Chapter 1

Neural network models

Let us begin with a simple illustration. A Facebook user uploads some image files containing photographs of family and friends. The system automatically highlights faces and is able to identify the names of many of the individuals in those photos. How? In fact, how does it even know what part of a complex, multifaceted image represents a human face? The short answer is, they use biometric data combined with specialized, deep learning algorithms based on artificial neural network (ANN) models. These algorithms attempt to mimic the current scientific understanding of how our human brain works, and how it learns new things.

In late 2021 Facebook announced the shutting down of its facial recognition feature, for a variety of socio-political reasons. But the technological innovations behind it continue to grow, and to power many other cutting-edge, real world applications, such as self-driving cars, voice recognition, credit card fraud detection, targeted advertising, and more. This chapter introduces the foundational concepts underlying ANNs and thier use in modern, data-centric applications. Our focus is primarily on methods that belong to the category known as "supervised learning" in machine learning nomenclature.

1.0.1 Conceptual overview

An artificial neural network (ANN) is essentially a mathematical model that receives a set of numeric inputs, based on which it produces an output. To a mathematician this, of course, describes a function, which is certainly one way to conceptualize an ANN. From a model-building perspective, an ANN is a set of very simple neuron-like components that are interconnected in the form of a network, whose structure and parameters determine what precise output is generated. It is common to think of the structure of these networks as comprised of layers (see Figure 1.1). Typically, there is an input layer and an output layer, together with optional layers in-between, usually referred to as hidden layers.
To understand the functioning of a neural network, let us look at how a single component, which we will call a neuron, works. Figure 1.2 shows a schematic of a neuron receiving n inputs and, following a sequence of steps, producing an output $\hat{y}$. Neurons typically take in multiple inputs and produce a single output. The effect of each input x_i is moderated by a numerical weight coefficient w_i, which represents the strength of the connection between x_i and the output. Typically, we want to compare the sum $\sum_{i=1}^{n} x_i \cdot w_i$ with some specified

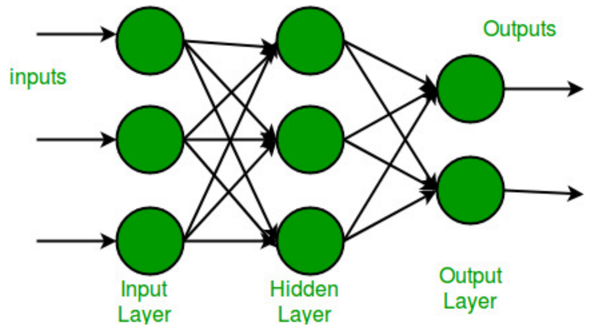


Figure 1.1: Example of layered structure of a typical neural network. [Image source: `https://www.geeksforgeeks.org/introduction-to-artificial-neural-networks/`]

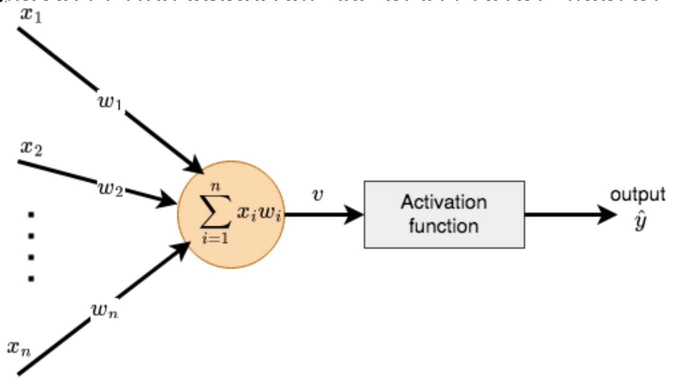


Figure 1.2: A neuron receiving n inputs and producing output $\hat{y}$.

threshold value b, known as the bias, to determine the neuron's output. This is done by an activation function that takes the input $\sum_{i=1}^{n} x_i \cdot w_i - b$ and produces the output $\hat{y}$. It is usual to absorb the effect of the bias within the summation by setting $x_0 = 1$ and $w_0 = -b$, so that $v = \sum_{i=0}^{n} x_i \cdot w_i$ is the input to the activation function. The output from the activation function represents that neuron's decision, or output. Commonly used activation functions include those that produce binary or discrete output, as well as others that produce continuous output within a specified range set (e.g., logistic function).

Example: The logical OR connective can be modeled as a very simple neural network. The truth table for the logical statement x_1 or x_2 is

x_1	0	0	1	1
x_2	0	1	0	1
$f(x_1, x_2)$	0	1	1	1

where 0 represents false and 1 represents true. The function $f(x_1, x_2)$ represents the OR operator. Fig. 1.3 shows the sketch of an ANN that implements this, with the activation function f being a unit step function. Thus, for example, if $x_1 = 0, x_2 = 1$, then $v = -1 + x_1 + x_2 = 0$ and $f(0) = 1$, which represents true.
This example, together with the preceding discussion, suggests that a neuron is basically a mechanism for weighing evidence to make a decision. The idea can be generalized to situa-

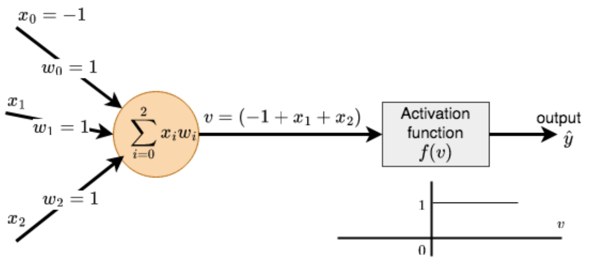


Figure 1.3: ANN schematic for logical OR connective.

tions with several inputs (or evidence types) being given to multiple neurons simultaneously, for making decisions about different aspects of the input. Each neuron performs a similar task, using its associated weights and activation function to produce its output. Collectively, this set of outputs comprises the final result, or it becomes the input for the next layer of neurons. In this way, each successive layer engages in more complex and abstract decision making, by weighing the results from the previous layer.

This type of information flow from the input layer, through hidden layers, to the output layer is called a feed forward process. In theory, it is possible to choose the number of neurons, layers, weights and activation functions to perfectly model almost any real-world system, no matter how complex. But, in practice it is challenging to find the right combination of network structure and parameters to do this. For this reason, a significant emphasis in ANN research has been on developing methods to find optimal weights and activation functions. This is where the training and learning aspect of neural networks comes in.

1.0.2 Training and learning methods

It is a common human experience to learn by practicing, making mistakes, and comparing one's attempt with the known, correct answer. This is the essense of supervised learning methods in machine learning. Supervised training and learning methods for ANNs are motivated by a similar idea. These methods typically employ an adaptive, iterative strategy for updating the weights in an ANN based on prediction errors. In order to do this, training data sets are used, where the answers that the ANN is supposed to find are known in advance. Then the errors in the predicted answers provide a concrete numerical basis for making improvements in the network's parameters.

Let us consider an example. Figure 1.4 shows a scatter plot of a dataset containing 1000 pairs of (x, y) values [1]. We want to model these data using an ANN that will predict the y value, given any input x value. Since the relationship looks approximately linear, we will build a 1-neuron network that models the relationship with a straight line. Readers familiar

[1]The ideas contained in this example are taken from the reference
`https://lucidar.me/en/neural-networks/linear-regression-example/`

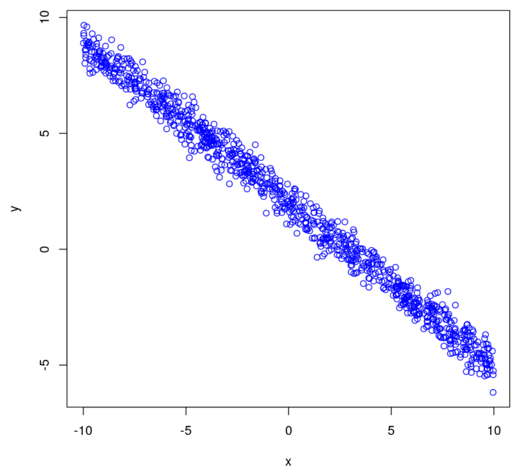


Figure 1.4: Scatter plot of dataset for ANN example.

with regression analysis will, of course, recognize this as the classic 1-variable linear regression problem. However, instead of using regression, we will train a neural netwrok to solve this problem. The training strategy we will use is known as *back propagation*.

The neuron receives one input (x), plus the bias, which is represented by the weight w_0. The goal is to find the values of the weights so that we get the most accurate predictions for y, the true output, which may or may not be known. The activation function is the identity, so the neuron outputs the value $\hat{y} = w_0 + w_1 x$, as its predicted output when x is the input.

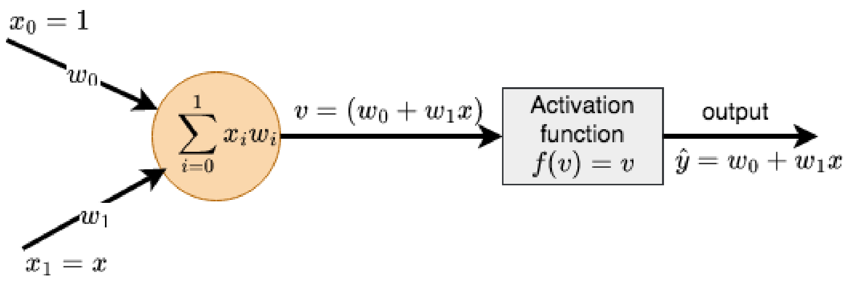


Figure 1.5: ANN schematic for regression example.

For training the network, we use the 1000 pairs of (x_i, y_i) values contained in the input dataset. The process starts with an initial guess for (w_0, w_1), say, $(w_0^{(0)}, w_1^{(0)})$, where superscripts denote the iteration number. This produces an initial model for the output $\hat{y}$, which is used with the first training data point (x_1, y_1) to estimate the prediction error, and to update the weights. Here are the steps

- Pick initial guess $w_0^{(0)}, w_1^{(0)}$

- The initial model is: $\hat{y} = w_0^{(0)} + w_1^{(0)} x$
- The prediction for input x_1 is: $\hat{y_1} = w_0^{(0)} + w_1^{(0)} x_1$
- The prediction error is: $y_1 - \hat{y_1}$ (note, we don't use absolute values here)
- Update the weights using: $\begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^{(1)} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^{(0)} + \delta \, (y_1 - \hat{y_1}) \begin{bmatrix} 1 \\ x_1 \end{bmatrix}$

  where δ is a constant learning rate parameter chosen in advance. Note that the input x_1 is turned into a vector with 1 as its first component to accommodate the bias, which is modeled by w_0.

This iterative process is carried out using all 1000 pairs of (x_i, y_i) values in the input dataset, and the update at the i^{th} step has the form

$$\begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^{(i+1)} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^{(i)} + \delta \, (y_{i+1} - \hat{y}_{i+1}) \begin{bmatrix} 1 \\ x_{i+1} \end{bmatrix} \tag{1.1}$$

Figure 1.6 shows the trained neural network model obtained for this example, after just one pass through the entire training data set. The final values of the weights are $w_0 = -0.712, w_1 = 2.12$, while the theoretical intercept and slope values are -0.7 and 2.0, respectively. An R code listing for this example is also provided below. Note that each execution of the code will produce somewhat different results, due to the use of random numbers for generating the input data.

Listing 1.1: R code for ANN example demonstrating linear regression

```
# Theoretical straight line parameters (line is: y=a+b*x)

a = 2.0       # intercept of str. line model
b = -0.7      # slope of str. line model

# Number of input data points to generate
n = 1000

# SD of normally distributed random noise to add to straight
   line
noisesd = 0.5

# Learning rate parameter of neural network
delta = 0.003

# Create (x, y) values of input dataset
xmodel = runif(n, -10, 10)
ymodel = a + b*xmodel + rnorm(n, mean=0, sd=noisesd)
```

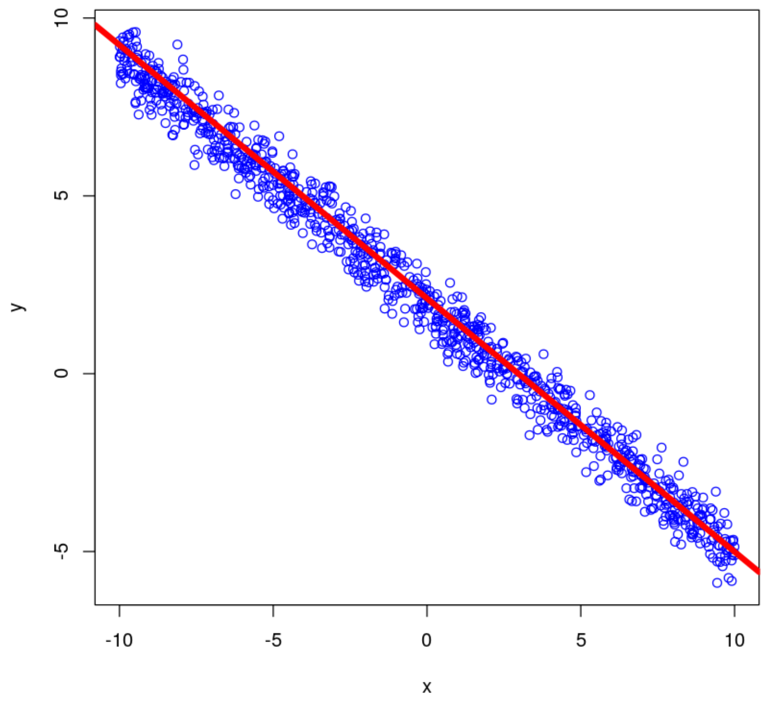


Figure 1.6: Trained neural network model (straight line in red) for regression example.

```
# Initial value of weight coefficients in neural network model:
    y = w1 + w2*x
w = c(0, 0)

# Loop over input/training data set
for ( i in 1:n ){
    y = w * c( 1, xmodel[i] )         # predicted output
    w = w + delta*(ymodel[i]-y)*c(1, xmodel[i])  # weight
        correction based on error
}


# Plot input/training data
plot(xmodel, ymodel, xlab="x", ylab="y", type="p", col="blue")

# Plot the model produced by the neural network
abline(w[1], w[2], col=2, lwd=5)

# Compare the theoretical model with that produced by the
```

```
   neural network
cat("\nThe␣theoretical␣model␣parameters␣are␣=", a, ",", b)
cat("\nThe␣optimized␣weights␣are␣=", w)
```

Although the training method used in the regression example appears to work satisfactorily, it will have raised several questions for the curious reader. In fact, here are some that we would like to ask:

1. How do we find the best value for the learning rate parameter (δ)? What happens if we pick different values for it?

2. How does the size of the training data set affect results?

3. What is the logic or justification for the formula we used for updating the weights?

To explore the answers to these questions, let us consider a more systematic approach to the objectives of training.

Training neural networks using optimization

One way to develop more reliable training methods is through the use of optimization strategies. Consider a general neural network consisting of a collection of weights and biases denoted by the vector $\mathbf{w}$. In the supervised learning approach, we want to find the best value of $\mathbf{w}$ such that the network's output matches the target output in the training set as closely as possible. Let us define this goal more precisely in the form of an average error or cost function

$$e(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1.2}$$

where subscript i corresponds to the i^{th} training input, $\hat{y}_i$ denotes the predicted output, y_i the expected (or target) output, and the summation is carried out over n training input sets. For clarity, we have assumed the network's output $\hat{y}_i$ is a single scalar quantity, though this strategy is not limited to that case. In the literature, this is often called the *Mean Squared Error* or *MSE*. Notice that $e(\mathbf{w})$ always has a non-negative value, and it is a function of $\mathbf{w}$ because each $\hat{y}_i$ depends on $\mathbf{w}$. We want to find the vector $\mathbf{w}$ that minimizes $e(\mathbf{w})$.

Optimization is an iterative process that starts from some initial guess, say $\mathbf{w}^{(0)}$. Each iteration involves two key conceptual steps: (1) find the optimal direction in which to move the current approximation $\mathbf{w}^{(k)}$, and (2) determine how much to move in that direction. Since the goal is to minimize $e(\mathbf{w})$, we seek to move in a direction that will reduce it. From mathematical theory, the gradient of $e(\mathbf{w})$ is the direction of maximum increase. Thus, we want to move in the direction of the negative gradient. This is the logic behind gradient descent methods, which are among the most popular training methods for ANNs.

To elaborate on some details, suppose $\mathbf{w} = (w_1, w_2, \ldots, w_M)^T$ in a network with M weights and biases. Then, the gradient vector is

$$\boldsymbol{\nabla} e = \left[\frac{\partial e}{\partial w_1}, \frac{\partial e}{\partial w_2}, \ldots, \frac{\partial e}{\partial w_M} \right]^T \tag{1.3}$$

The update for the weight vector going from iteration k to $k+1$ is then

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \delta \; \boldsymbol{\nabla} e(\mathbf{w}^{(k)}) \tag{1.4}$$

where δ is a constant and $\boldsymbol{\nabla} e$ is evaluated at $\mathbf{w}^{(k)}$. Since the negative gradient is a direction that decreases e, this update is guaranteed to reduce the prediction error, provided δ is small enough.

In practice, computing the derivatives in $\boldsymbol{\nabla} e$ typically involves application of the chain rule. To elucidate, let us revisit the linear regression example in Figures 1.5-1.6, and consider just a single training input. The *MSE* is

$$e(\mathbf{w}) = \frac{1}{2}(y - \hat{y})^2 \tag{1.5}$$

The vector of weights in that example is $\mathbf{w} = (w_0, w_1)^T$, and the output is given by $\hat{y} = w_0 + w_1 x$. Therefore,

$$\boldsymbol{\nabla} e = \begin{bmatrix} \dfrac{\partial e}{\partial w_0} \\ \dfrac{\partial e}{\partial w_1} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial e}{\partial \hat{y}} \cdot \dfrac{\partial \hat{y}}{\partial w_0} \\ \dfrac{\partial e}{\partial \hat{y}} \cdot \dfrac{\partial \hat{y}}{\partial w_1} \end{bmatrix} = \begin{bmatrix} -(y - \hat{y}) \cdot 1 \\ -(y - \hat{y}) \cdot x \end{bmatrix} = -(y - \hat{y}) \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{1.6}$$

This gives the following formula to update the weights

$$\begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^{new} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}^{old} + \delta \; (y - \hat{y}) \begin{bmatrix} 1 \\ x \end{bmatrix} \tag{1.7}$$

which is exactly the same as the formula we used [see equation (1.1)]. Thus, the training method used in the regression example is, in fact, based on a gradient descent algorithm. And, the numerical value of δ controls how far the current iterate moves in the direction of the negative gradient. Ideally, we want the value of δ to be such that the new iterate $\mathbf{w}$ minimizes $e(\mathbf{w})$. In our example, δ was chosen based on heuristics, and it was not necessarily the best value for it.

A clear understanding of the back propagation process with gradient descent optimization is extremely helpful for generalizing these ideas to multilayer networks. Let us consider one more relatively simple example. Earlier in this chapter we looked at the logical "or" connective, and learned how to build a 1-neuron model for it, with appropriate weights and bias. Suppose we consider the exact same problem, except this time we won't tell the network what values to use for the weights and bias. We will train the network to figure out the values on its own.

Figure 1.7 shows the schematic and notation we use, which will be helpful for interpreting key steps in the R code given below. Note that x_3 and w_3 represent the bias in this example, in order to retain the intuitive interpretation of x_1, x_2 as inputs to the OR connective.

x_1	x_2	x_1 or x_2
0	0	0
0	1	1
1	0	1
1	1	1

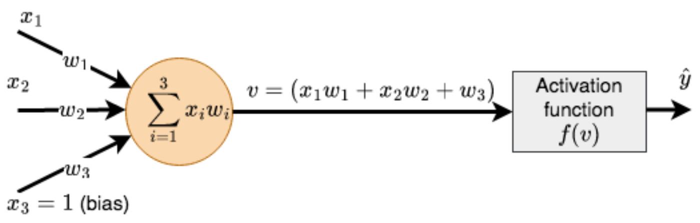


Figure 1.7: Training a 1-neuron network for the logical or connective.

Our training data set consists of the four possible combinations of input, and the corresponding outputs, as shown in the table. The activation function f is the unit step, with threshold at 0. The inputs and weights are represented by the vectors $\mathbf{x} = (x_1, x_2, x_3)^T$, $\mathbf{w} = (w_1, w_2, w_3)^T$ respectively. The predicted output is $\hat{y} = f(\mathbf{x} \cdot \mathbf{w})$. We use the error indicator $e(\mathbf{w}) = \frac{1}{2}(y - \hat{y})^2$, whose gradient is $\boldsymbol{\nabla} e = -(y - \hat{y})\ f'(\mathbf{x} \cdot \mathbf{w})\ \mathbf{x}$. Thus, the update formula for the weights is

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \delta(y - \hat{y}) f'(\mathbf{x} \cdot \mathbf{w}^{(\mathbf{k})})\mathbf{x}$$

where $(y - \hat{y})$ is the difference between the desired and the predicted output when the input is $\mathbf{x}$. The learning rate δ will be chosen based on heuristics, and we will investigate its effect on the results. Note that we actually run into a problem here, since the unit step function yields $f'(\mathbf{x} \cdot \mathbf{w}) = 0$ for all $\mathbf{x} \cdot \mathbf{w} \neq 0$, which means the updates to the weights will be 0. Nevertheless, let us proceed by setting it to a non-zero constant (say, 1) and observe the effect.

In the R code given below, we implement this process by looping through each of the 4 pairs of training inputs, and updating the weights. This completes one *epoch*, the technical term for one pass through the full training data set. Subsequent epochs will cycle through the training data again and repeat the above steps. The code allows the user to set the value of the following parameters, and we invite the reader to explore their effect on the results:

- The learning rate parameter δ.
- Number of epochs.
- Threshold input value that triggers the activation function.
- Initial guess for the weights.

In the sample results shown, we used $\delta = 0.03$, with 5 epochs, an activation threshold of 0, and all initial weights set to 0. This produces the result $\mathbf{w} = (0.03, 0.03, -0.03)^T$, which the code also tests and verifies that it works correctly. Clearly, the result is not unique, and other values of δ and/or activation thresholds will yield different weights that also work correctly.

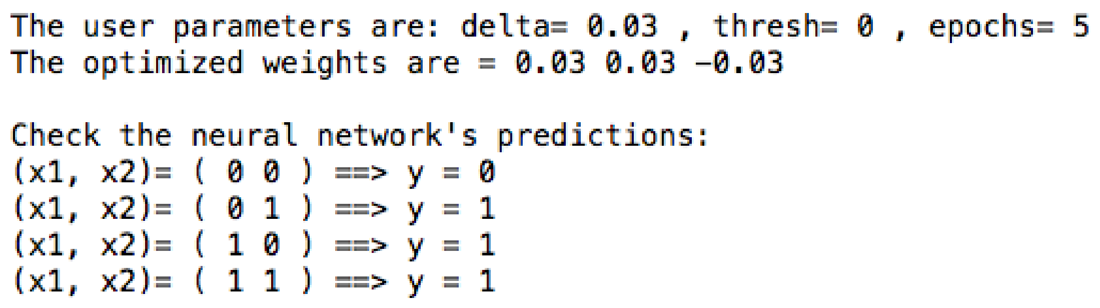
Remark: The R and Python codes we provide in this chapter are primarily intended to illustrate the underlying methods, and to implement the solution approaches discussed in our examples. These codes work well for our purpose, but they are not written to be elegant, efficient or flexible. There are several reputable and powerful open-source libraries and packages available for implementing more general neural network models. These packages provide enormous flexibility and efficiency in terms of network structure, training methods, and more. We will briefly discuss some of these at the end of this chapter.

Listing 1.2: R code for training a 1-neuron network for the logical or connective

```
# Example: Training a 1-neuron neural network to simulate
# logical OR connective.
#
# Inputs to the neural network: x1, x2 (binary values 0 or 1)
# Output from the network: x1 OR x2 (binary)
#
# User parameters: delta, thresh, epochs, w (initial value)
# delta = learning rate of neurons
# thresh = activation threshold for unit step function
# epochs = number of cycles over training data
# w = value of the three weight parameters (2 weights + 1 bias)


# Set the learning rate parameter of neural network
delta = 0.03

# Set the threshold parameter of activation function
thresh = 0

# Set the number of epochs (cycles over training data)
epochs = 5

# Create (x, y) values of input dataset: For logical OR
   connective
x1 = c(0, 0, 1, 1)
x2 = c(0, 1, 0, 1)
ymodel = c(0, 1, 1, 1)

# Set initial value of weight coefficients
```

```
w = c(0, 0, 0)


# Loop over the number of epochs (how many passes over training
    data)
for ( k in 1:epochs ){

# Loop over input/training data set
    for ( i in 1:4 ){
        v = w[1]*x1[i]+w[2]*x2[i]+w[3]
        y = 0
        if (v >= thresh) {
            y = 1
        }

        w = w + delta*(ymodel[i]-y)*c( x1[i], x2[i], 1 )  #
           weight correction based on error
    }
}

# Compare the theoretical model with that produced by the
   neural network
cat("\nThe␣user␣parameters␣are:␣delta=", delta, ",␣thresh=",
   thresh, ",␣epochs=", epochs)
cat("\nThe␣optimized␣weights␣are␣=", w)
cat("\n\nCheck␣the␣neural␣network's␣predictions:")
for ( i in 1:4 ){
    yout = 0
    v=w[1]*x1[i]+w[2]*x2[i]+w[3]
    if (v >= thresh) {
        yout = 1
    }
    cat("\n(x1,␣x2)=", "(", x1[i], x2[i], ")", "==>␣y␣=", yout)
}
```

Here is the output from running the code



```
The user parameters are: delta= 0.03 , thresh= 0 , epochs= 5
The optimized weights are = 0.03 0.03 -0.03

Check the neural network's predictions:
(x1, x2)= ( 0 0 ) ==> y = 0
(x1, x2)= ( 0 1 ) ==> y = 1
(x1, x2)= ( 1 0 ) ==> y = 1
(x1, x2)= ( 1 1 ) ==> y = 1
```

Thus, the trained network does produce correct results for all the cases.

Multilayer networks

We are now ready to discuss more advanced neural network structures that involve multiple layers, and more general activation functions. As a first example, let us look at how to train a neural network to mimic nonlinear regression. Figure 1.8 shows a scatter plot of an x-y relationship that is clearly unsuitable for a straight line approximation.

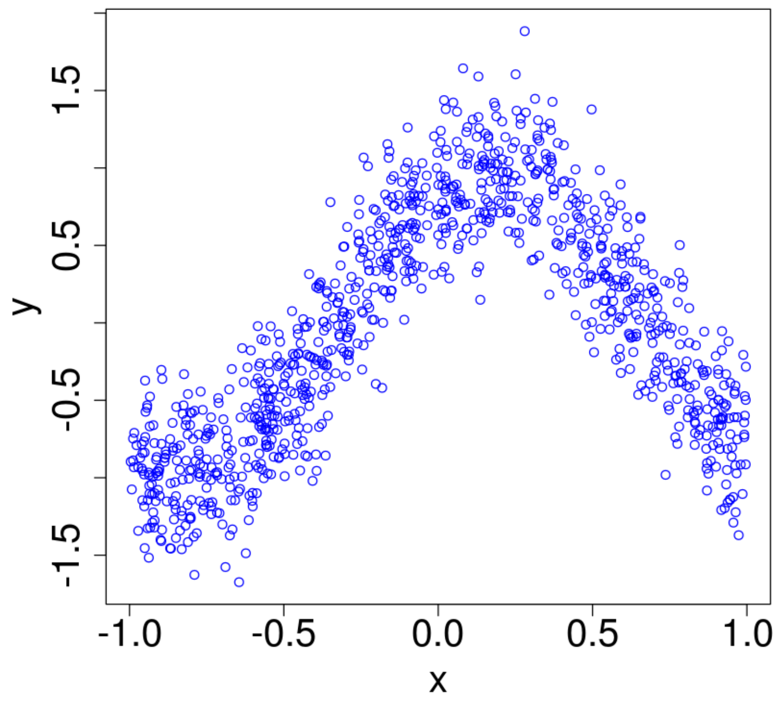


Figure 1.8: Scatter plot for nonlinear regression example.

We will build a neural network consisting of the structure shown in Figure 1.9 to model this relationship. The single input neuron is followed by a hidden layer consisting of two neurons, followed by an output layer with one neuron. The input neuron simply transmits the x input to the next layer, which contains two neurons h_1 and h_2. Each of thse neurons behaves exactly like a 1-neuron case: linearly combine inputs, apply activation function, transmit output to next layer. The output layer contains one neuron, whose behavior is similar, and it produces the final result $\hat{y}$. For clarity, the three activation functions are not shown in the sketch. All three activation functions will be taken to be the hyperbolic tangent $f(x) = \tanh(x)$, which is continuous and differentable everywhere. Its output is in the range $(-1, 1)$ and its derivative is $f'(x) = 1 - \tanh^2(x)$. See Table 1 for graphs of this and other common activation functions.

As shown in the sketch, there are a total of 7 weights (including the biases) whose optimal values we want to determine by training the network. We will use gradient descent to

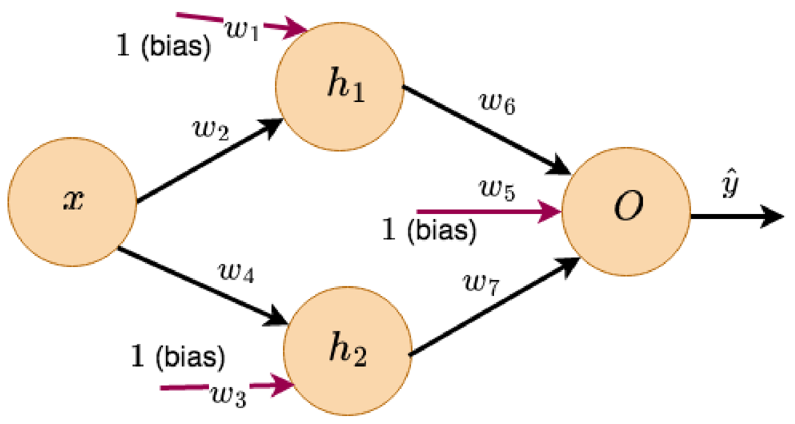


Figure 1.9: Neural network structure for nonlinear regression example.

minmize the error defined by

$$e(\mathbf{w}) = \frac{1}{2}(y - \hat{y})^2$$

where $(y - \hat{y})$ is the difference between the target output and the predicted output for any given x-value, and $\mathbf{w}$ represents the vector of weights and biases. Despite the relative simplicity of this neural network, a careful derivation of the gradients used in the back propagation algorithm is extremely insightful. Accordingly, we have

$$\boldsymbol{\nabla} e = -(y - \hat{y})\ \boldsymbol{\nabla}\hat{y}$$

To compute $\boldsymbol{\nabla}\hat{y}$, we need to know the relationship between $\hat{y}$ and the 7 components of $\mathbf{w}$. This is easiset to see with the help of Figure 1.9 by working backward from $\hat{y}$, which gives

$$\begin{aligned} \hat{y} &= \tanh(z) \\ z &= w_5 + w_6 h_1 + w_7 h_2 \\ h_1 &= \tanh(u) \\ h_2 &= \tanh(v) \\ u &= w_1 + w_2 x \\ v &= w_3 + w_4 x \end{aligned} \tag{1.8}$$

The intermediate variables $(z, h_1, u, \ldots,$ etc.$)$ are introduced to elucidate the role of each neuron, as well as to help in computing the gradient. The components of $\boldsymbol{\nabla}\hat{y}$ can be found using the chain rule as follows

$$\begin{aligned}
\frac{\partial \hat{y}}{\partial z} &= \tanh'(z) \\
\frac{\partial \hat{y}}{\partial w_7} &= \frac{\partial \hat{y}}{\partial z}\frac{\partial z}{\partial w_7} = \tanh'(z)h_2 \\
\frac{\partial \hat{y}}{\partial w_6} &= \tanh'(z)h_1 \\
\frac{\partial \hat{y}}{\partial w_5} &= \tanh'(z) \\
\frac{\partial \hat{y}}{\partial w_4} &= \frac{\partial \hat{y}}{\partial z}\cdot\frac{\partial z}{\partial h_2}\cdot\frac{\partial h_2}{\partial v}\cdot\frac{\partial v}{\partial w_4} = \tanh'(z)\cdot w_7\cdot\tanh'(v)\cdot x \\
\frac{\partial \hat{y}}{\partial w_3} &= \tanh'(z)\cdot w_7\cdot\tanh'(v)\cdot 1 \qquad (1.9)\\
\frac{\partial \hat{y}}{\partial w_2} &= \frac{\partial \hat{y}}{\partial z}\cdot\frac{\partial z}{\partial h_1}\cdot\frac{\partial h_1}{\partial u}\cdot\frac{\partial u}{\partial w_2} = \tanh'(z)\cdot w_6\cdot\tanh'(u)\cdot x \\
\frac{\partial \hat{y}}{\partial w_1} &= \tanh'(z)\cdot w_6\cdot\tanh'(u)\cdot 1
\end{aligned}$$

Once the gradients are computed, the weights are updated like usual

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \delta(y-\hat{y})\boldsymbol{\nabla}\hat{y}$$

where δ is the learning rate parameter. In the R code given below, we implement this algorithm by looping over the entire input set of x values several times, indicated by the variable `epochs`. Other user customizable parameters in the code are δ, and the initial values of the 7 weights and biases. Figure 1.10 shows sample results, including the original scatter plot, together with the output predicted by the trained network for a random set of 200 x values. These results were obtained starting from randomized initial weights (between -0.5 and 0.5), and with $\delta = 0.1$, `epochs`=20. The values obtained for the weights after 20 epochs are: $w_1 = 1.9009, w_2 = -3.3486, w_3 = 0.7668, w_4 = 2.4180, w_5 = -1.4468, w_6 = 1.4934, w_7 = 1.8030$. Note, however, that these results are not unique, and each execution of the same code will produce a different set of optimized weights, due to the use of randomization in the input data and in the initial weights.

Listing 1.3: R code for training a multilayer network to mimic nonlinear regression

```
# This example illustrates the use of a (simple) multilayer
# neural network to mimic nonlinear regression.
# We will first create a data set of a strongly nonlinear
# relationship between two variables (x,y).  Then we will train
# our neural network using that data set.

# The structure of our neural network consists of a single
  input
# neuron, followed by a hidden layer consisting of two neurons,
```

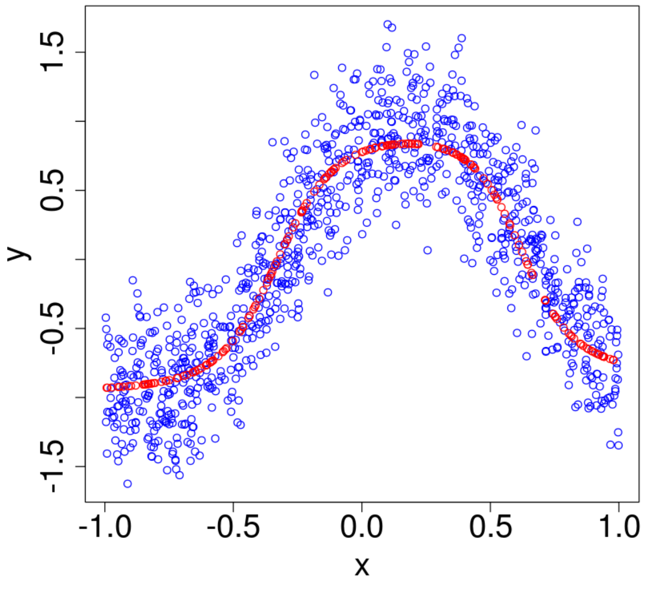


Figure 1.10: Trained neural network's prediction (in red) for nonlinear regression example.

```
# followed by an output layer with one neuron.
# We will use the activation function: f(x)=tanh(x)
# There are a total of 7 weights (including the biases) whose
# optimal values we want to determine by training the network.
# We will use the gradient descent method for training.

#--------------- The code starts here -----------------------
# Create training data.  We use a sinusoidal curve with
# random noise added.
# Original sinusoid model parameters (sinusoidal curve is: y=a+
   b*sin(pi*x+c))

a = 0
b = 1
c = 1

# Number of input/training data points to generate:
n = 1000

# SD of normally distributed random noise to add to curve:
noisesd = 0.3

# Learning rate parameter of neural network:
delta = 0.05
```

```
# Create (x, y) values of input dataset on domain (-1, 1):
xmodel = runif(n, -1, 1)
ymodel = a + b*sin(pi*xmodel+c) + rnorm(n, mean=0, sd=noisesd)


# Initial value of weights: randomized, between -0.5 and 0.5
#w = c(0, 0, 0, 0, 0, 0, 0)
w = runif(7, -0.5, 0.5)


# Network structure and relationship of output (y) to weights:
#
# input neuron --> x --> hidden neuron1 --> u = w1 + w2*x
# input neuron --> x --> hidden neuron2 --> v = w3 + w4*x
# f(u), f(v) --> output neuron --> z = w5 + w6*f(u) + w7*f(v)
# predicted output: y = f(z).
# The activation function is: f(x)=tanh(x)  and f'(x) = 1-(tanh
   (x))^2

# Gradient components (via chain rule):
# dy/dz = f'(z)
# dy/dw7 = f'(z) * f(v)
# dy/dw6 = f'(z) * f(u)
# dy/dw5 = f'(z)
# dy/dw4 = f'(z) * w7 * f'(v) * x
# dy/dw3 = f'(z) * w7 * f'(v)
# dy/dw2 = f'(z) * w6 * f'(u) * x
# dy/dw1 = f'(z) * w6 * f'(u)


# Loop over the number of epochs (how many passes over training
    data):
epochs = 20
for ( k in 1:epochs ){

# Loop over input/training data set:
    for ( i in 1:n ){

# Compute predicted output:
        u = w[1]+w[2]*xmodel[i]
        v = w[3]+w[4]*xmodel[i]
        z = w[5]+w[6]*tanh(u)+w[7]*tanh(v)
        y = tanh(z)

# Compute gradient components:
        dydz = 1-(tanh(z))^2
        dzdv = w[7]*(1-(tanh(v))^2)
        dzdu = w[6]*(1-(tanh(u))^2)

        dydw1 = (dydz)*(dzdu)
        dydw2 = (dydz)*(dzdu)*xmodel[i]
        dydw3 = (dydz)*(dzdv)
        dydw4 = (dydz)*(dzdv)*xmodel[i]
        dydw5 = dydz
        dydw6 = dydz*tanh(u)
        dydw7 = dydz*tanh(v)

# Update weights:
        w = w + delta*(ymodel[i]-y)*c(dydw1, dydw2, dydw3,
           dydw4, dydw5, dydw6, dydw7)  # weight correction
           based on error
    }
}


# Plot input/training data:
plot(xmodel, ymodel, xlab="x", ylab="y", type="p", col="blue",
     cex.lab=2, cex.axis=2)

# Plot the model produced by the neural network:
nverify = 200
xverify = runif(nverify, -1, 1)
yverify = 0*xverify
for ( i in 1:nverify ){
    u = w[1]+w[2]*xverify[i]
    v = w[3]+w[4]*xverify[i]
    z = w[5]+w[6]*tanh(u)+w[7]*tanh(v)
    yverify[i] = tanh(z)
    }
points(xverify, yverify, type="p", col="#FF0000")

# Print the values obtained for the weights:
cat("\nThe␣optimized␣weights␣are␣=", w)
```

This example provides a glimpse into the potential power and relative simplicity of neural network models. We provided the model with nothing but a random data set, and asked it to figure out an approximate relationship to capture any discernable pattern in the data. This is in striking contrast to regression modeling, where the user must provide a specific function form to fit the data. Another point this example illustrates is the plausibility of using neural network models for feature identification in more advanced real-world applications, such as face recognition, hand-writing recognition, etc. Such applications, essentially, involve recognizing patterns in data sets, which seems to be something we can train neural networks to do well.

Of course, at this point the alert reader will likely have several questions that we have not yet answered. Some of them might be

- How do we determine what structure to use for our neural network? In other words, how many layers should we use? How many neurons in each layer?
- Why did we pick the hyperbolic tangent for our activation function? Would other activation functions work as well?
- How important is the initial guess for the weights? Why did we set them to 0 in one example, and randomize them in the next example?
- What about δ and the number of `epochs`? Are there strategies or guidelines for how to choose them?

Unfortunately, not all of these questions have known answers, but we will provide some pointers towards what is currently known. On the question of network structure, two points should be clear: (a) It is not unique; and (b) it strongly depends on the underlying application. For instance, in the nonlinear regression example we could have used three (or more) neurons in the hidden layer, or we could have used more hidden layers. But, since the number of unknown parameters and the computational complexity increase with the number of neurons, we opted for the minimum structure that seemed likely to work. In a more general application setting, it may not be known what is a reasonable minimum number of neurons or hidden layers to use. In such situations it is usual to rely on some combination of heuristics and overkill. That means, a large number of neurons and layers is used, in conjunction with some trial and error, to arrive at a structure that does the job. The number of input and output neurons is often guided by the application, as was the case in our examples.

On the question of activation function, it turns out the choice is not too critical, as long as the function's behavior is within range of what is typical. Activation functions try to mimic the behavior of brain neurons, in that the neuron remains dormant until it receives an input signal strong enough to trigger a response. For mathematical neurons, an additional property that is desirable is differentiability. Thus, we typically want differentiable functions with domain $(-\infty, \infty)$, and with bounded, monotonic range. Table 1.1 shows some of the commonly used activation functions. We will illustrate the use of the sigmoid function for activation in a later example.

The initial guess for the weights can make a difference, both from the standpoint of whether the solution converges at all, as well as the speed at which it converges. In our experience, setting all the initial weights to 0, or to any single common value, is not a good idea. We

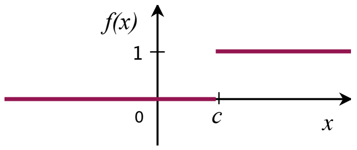
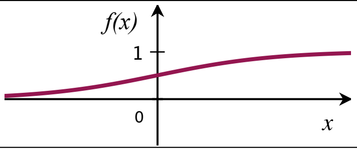
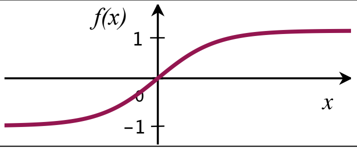
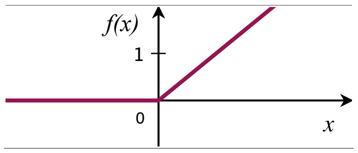
Function	Algebraic form	Graph	Derivative
Threshold/ unit step	$f(x) = \begin{cases} 0 & \text{if } x < c \\ 1 & \text{if } x \geq c \end{cases}$ for constant c		$f'(x) = \begin{cases} 0 & \text{if } x \neq c \\ \text{DNE} & \text{if } x = c \end{cases}$
Sigmoid/ logistic	$f(x) = \dfrac{1}{1+e^{-x}}$		$f'(x) = f(x)\,(1 - f(x))$
Hyperbolic tangent	$f(x) = \tanh(x)$		$f'(x) = 1 - \tanh^2(x)$
ReLU	$f(x) = \max(0, x)$		$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{DNE} & \text{if } x = 0 \end{cases}$

Table 1.1: Common activation functions and their derivatives.

found behaviors ranging from slow convergence, to non-convergence with such initial guesses. Randomized values for the initial weights seems to produce more robust convergence. However, we encourage the reader to explore this question further by trying different strategies in the examples and exercises of this chapter. In more advanced applications on large networks (with many more weights and biases), better strategies for initializations do exist. We will discuss these briefly later in this chapter.

Methods for choosing the value of δ and the number of epochs is typically guided by experience, and by an understanding of the role they play in the training process. In the gradient descent method, δ controls how far the current approximation is moved in the direction of the negative gradient. Moving too little will mean we don't get the full benefit on each iteration that is attainable. This will result in slower convergence. Moving too much will cause overshoot past the point of reducing the cost function, for which the algorithm must compensate in later iterations. This will also slow the convergence, and in some cases even lead to divergence. We recommend starting with a conservatively small δ, and experimenting with the effect of increasing it. More advanced strategies for δ involve changing its value in some adaptive fashion as the convergence progresses.

As for the number of epochs, one reasonable option is to monitor the magnitude of the cost function, and to exit when its value (or the change in its value) drops below some specified tolerence. In this case, the user parameter `epochs` serves as a ceiling or upper limit. The actual number of epochs will be determined automatically by the algorithm. A word of caution, particularly for large applications with many weights and biases, is to beware of a phenomenon called *overfitting*. This basically occurs when a neural network has so many free parameters that it is possible to fit the training data extremely well, and still be a poor model for the underlying application. We will discuss overfitting in a little more detail later.

Now that have a bigger picture view, let us consider one more example.

Example: The logical XOR operator offers another interesting case study in the use of multilayer networks. As a warmup exercise, we invite the reader to modify the R code we used for the OR connective and try to train the 1-neuron network to model XOR, which returns 1 (or true) when exactly one of the inputs is 1, as shown in this table

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Unlike the OR operator, the domain of the XOR operator is not linearly separable. In other words, there is no straight line that that will divide the domain into two groups of input that correctly map to the two corresponding outputs. A 1-neuron network can only model situations that are linearly separable. Thus, we will need a multilayer network to model the XOR operator.

Figure 1.11 shows a schematic of a relatively simple multilayer ANN that we will attempt to train for modeling XOR. Notice this structure is almost the same as the one we used for the nonlinear regression example, except here we need 2 input neurons, since the XOR operator receives 2 inputs. As before, neurons h_1, h_2 and O each have an activation function that is not shown in the sketch. All of them will be taken to be the sigmoid function, which is defined by

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Like the hyperbolic tangent, the sigmoid function produces a continuous output and is differentable everywhere, making it suitable for applying gradient descent strategies. It is worth noting that despite the similarity in the ANN structure, this problem belongs to a completely different class than the regression problem. The XOR operator's output is binary (yes or no), which essentially makes it a classification type of problem. On the other hand, the regression problem produces a continuous numeric output in $\mathbb{R}$. Yet, it is remarkable that almost the same neural network strategy can be used for both problem types.
For training the network, we must find optimal values of the 6 weights and 3 biases shown in the sketch. Note that the symbols denoting the weights are: $w_{11}, w_{12}, w_{21}, w_{22}, o_1, o_2$. We will use gradient descent to minmize the error defined by

$$e(\mathbf{w}) = \frac{1}{2}(y - \hat{y})^2$$

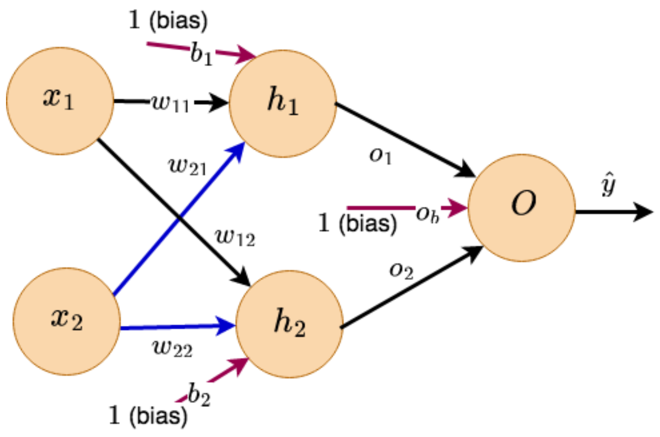


Figure 1.11: ANN schematic for modeling the XOR connective.

where $(y - \hat{y})$ is the difference between the desired and the predicted output for any given input, and $\mathbf{w}$ represents the vector of weights and biases. Thus, the gradient is

$$\boldsymbol{\nabla} e = -(y - \hat{y})\ \boldsymbol{\nabla} \hat{y}$$

To compute $\boldsymbol{\nabla} \hat{y}$, we must define the relationship between $\hat{y}$ and the 9 components of $\mathbf{w}$. With the help of the sketch, working backward from $\hat{y}$, we have

$$\begin{aligned}
\hat{y} &= \sigma(z) \\
z &= o_1 h_1 + o_2 h_2 + o_b \\
h_1 &= \sigma(v_1) \\
h_2 &= \sigma(v_2) \\
v_1 &= w_{11} x_1 + w_{21} x_2 + b_1 \\
v_2 &= w_{12} x_1 + w_{22} x_2 + b_2
\end{aligned} \tag{1.10}$$

Thus, the components of $\boldsymbol{\nabla}\hat{y}$ can be found using the chain rule as follows

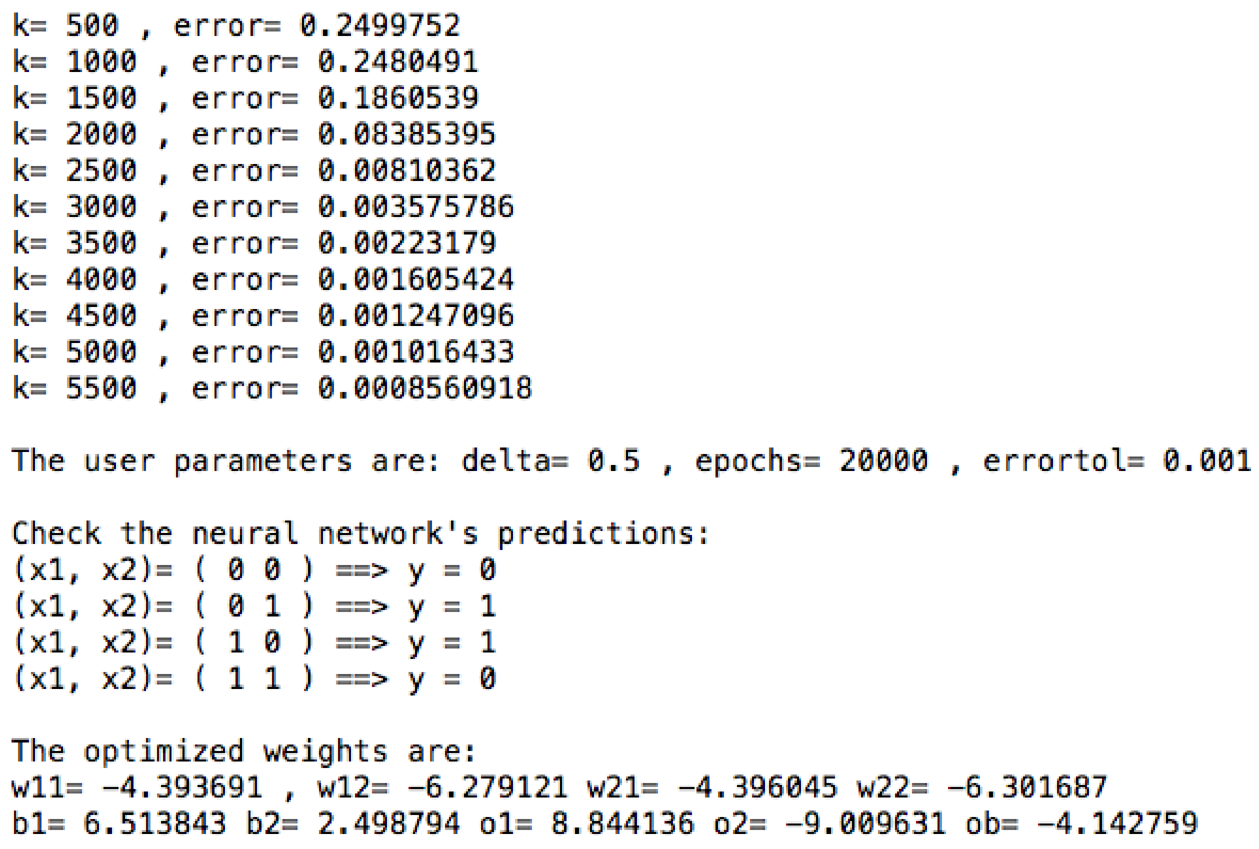
$$\begin{aligned}
\frac{\partial \hat{y}}{\partial z} &= \sigma'(z) \\
\frac{\partial \hat{y}}{\partial o_1} &= \frac{\partial \hat{y}}{\partial z}\,\frac{\partial z}{\partial o_1} = \sigma'(z) h_1 \\
\frac{\partial \hat{y}}{\partial o_2} &= \sigma'(z) h_2 \\
\frac{\partial \hat{y}}{\partial o_b} &= \sigma'(z) \\
\frac{\partial \hat{y}}{\partial w_{11}} &= \frac{\partial \hat{y}}{\partial z}\cdot\frac{\partial z}{\partial h_1}\cdot\frac{\partial h_1}{\partial v_1}\cdot\frac{\partial v_1}{\partial w_{11}} = \sigma'(z)\cdot o_1\cdot\sigma'(v_1)\cdot x_1 \\
\frac{\partial \hat{y}}{\partial w_{21}} &= \sigma'(z)\cdot o_1\cdot\sigma'(v_1)\cdot x_2 \qquad (1.11) \\
\frac{\partial \hat{y}}{\partial b_1} &= \sigma'(z)\cdot o_1\cdot\sigma'(v_1)\cdot 1 \\
\frac{\partial \hat{y}}{\partial w_{12}} &= \frac{\partial \hat{y}}{\partial z}\cdot\frac{\partial z}{\partial h_2}\cdot\frac{\partial h_2}{\partial v_2}\cdot\frac{\partial v_2}{\partial w_{12}} = \sigma'(z)\cdot o_2\cdot\sigma'(v_2)\cdot x_1 \\
\frac{\partial \hat{y}}{\partial w_{22}} &= \sigma'(z)\cdot o_2\cdot\sigma'(v_2)\cdot x_2 \\
\frac{\partial \hat{y}}{\partial b_2} &= \sigma'(z)\cdot o_2\cdot\sigma'(v_2)\cdot 1
\end{aligned}$$

The derivative of the sigmoid function is straightforward to find: $\sigma'(z) = \sigma(z)[1-\sigma(z)]$. Once the gradients are computed, the weights are updated like usual

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \delta(y-\hat{y})\boldsymbol{\nabla}\hat{y}$$

where δ is the learning rate parameter. In the R code given below, we implement this algorithm by looping over the 4 combinations of (x_1, x_2) inputs several times, indicated by the variable `epochs`. Other user customizable parameters in the code are δ, the initial values of the 9 weights and biases, an optional error tolerance (`errtol`), and a parameter that sets the frequency of diagnostic output (`errorprintout`).

In the sample results shown, we used $\delta = 0.5$, with the maximum number of epochs set to 20,000, and `errortol`$=10^{-3}$. The initial values of the weights were uniformly random between -0.1 and 0.1. With these choices, we found the algorithm typically converges and exits after about 5,000-6,000 epochs. Here is a typical output from running the code



```
k= 500 , error= 0.2499752
k= 1000 , error= 0.2480491
k= 1500 , error= 0.1860539
k= 2000 , error= 0.08385395
k= 2500 , error= 0.00810362
k= 3000 , error= 0.003575786
k= 3500 , error= 0.00223179
k= 4000 , error= 0.001605424
k= 4500 , error= 0.001247096
k= 5000 , error= 0.001016433
k= 5500 , error= 0.0008560918

The user parameters are: delta= 0.5 , epochs= 20000 , errortol= 0.001

Check the neural network's predictions:
(x1, x2)= ( 0 0 ) ==> y = 0
(x1, x2)= ( 0 1 ) ==> y = 1
(x1, x2)= ( 1 0 ) ==> y = 1
(x1, x2)= ( 1 1 ) ==> y = 0

The optimized weights are:
w11= -4.393691 , w12= -6.279121 w21= -4.396045 w22= -6.301687
b1= 6.513843 b2= 2.498794 o1= 8.844136 o2= -9.009631 ob= -4.142759
```

Listing 1.4: R code for training a multilayer network to model the XOR connective

```
# Example: Training a multilayer neural network to simulate
# logical XOR connective, whose I/O looks like
#
#         x1    x2    x1 XOR x2
#          0     0        0
#          0     1        1
#          1     0        1
#          1     1        0
#
# We will train our neural network using this data set.
#
# The structure of our neural network consists of two input
# neurons, followed by a hidden layer consisting of two neurons
  ,
# followed by an output layer with one neuron.
# We will use the activation function: f(x)=sigmoid(x)
# There are a total of 9 weights (including the biases) whose
# optimal values we want to determine by training the network.
# We will use the gradient descent method for training.
#
# Inputs to the neural network: x1, x2 (binary values 0 or 1)
# Output from the network: x1 XOR x2 (continuous, between 0 and
    1)
# NOTE: The actual computed output is a continuous value
# between 0 and 1, but we round it to 0 or 1 at the end.
#
# User parameters: delta, epochs, w, errorprintout, errortol
# delta = learning rate of neurons
# epochs = number of cycles over training data
# w = initial values of the 9 weight parameters (6 weights + 3
   biases)
# errorprintout = compute and print MSE every "errorprintout"
   epochs
# errortol = break and exit epochs if MSE < errortol
#
#--------------- The code starts here -----------------------
#
# Define sigmoid function:
#
sigmoidfunc = function (x) {
    out = 1 / (1+exp(-x))
    return (out)
}

# Define sigmoid's derivative, given an input that is already
   sigmoided:
#
dsigmoid = function (z) {
# Assumes z = 1 / (1+exp(-x)), and we want dz/dx
    return (z*(1-z))
}

# Define function to predict output of neural network, given x1
   , x2 inputs.
# Note, this assumes the weights are available as global
   variables.
#
predict = function (x1, x2) {
    h1 <<- w11*x1 + w21*x2 + b1
    h1 <<- sigmoidfunc(h1)
    h2 <<- w12*x1 + w22*x2 + b2
    h2 <<- sigmoidfunc(h2)
    z = h1*o1 + h2*o2 + ob
    yhat = sigmoidfunc(z)
    return (yhat)
}

# Define function to train the network, given x1, x2 inputs,
   and
# desired/target output.  Also takes in delta as input.
#
train = function (x1, x2, target, delta) {
    y = predict (x1, x2)
    error = target - y

    dydz = dsigmoid(y)

    dydo1 = dydz * h1
    dydo2 = dydz * h2
    dydob = dydz

    dzdh1 = o1*dsigmoid(h1)
    dydh1 = dydz * dzdh1
    dydh2 = dydz * o2*dsigmoid(h2)

    dydw11 = dydh1 * x1
    dydw21 = dydh1 * x2
    dydb1 = dydh1
    dydw12 = dydh2 * x1
    dydw22 = dydh2 * x2
    dydb2 = dydh2

    o1 <<- o1 + delta * error * dydo1
    o2 <<- o2 + delta * error * dydo2
    ob <<- ob + delta * error * dydob

    w11 <<- w11 + delta * error * dydw11
    w21 <<- w21 + delta * error * dydw21
    b1 <<- b1 + delta * error * dydb1
    w12 <<- w12 + delta * error * dydw12
    w22 <<- w22 + delta * error * dydw22
    b2 <<- b2 + delta * error * dydb2

}

# The main program starts here.  It sets the user parameters,
   and
# controls the execution sequence, together with outputing
```

```
   results.

# Set the learning rate parameter:
delta = 0.5

# Set the number of epochs, and error printout criteria:
epochs = 20000
errorprintout = 500
errortol = 0.001; mse = 1

# Create (x, y) values of input dataset: For logical XOR
   connective
x1 = c(0, 0, 1, 1)
x2 = c(0, 1, 0, 1)
ymodel = c(0, 1, 1, 0)

# Set initial values of weight coefficients:
w = runif(9, -0.1, 0.1)
w11 = w[1]; w12 = w[2]; w21 = w[3]; w22 = w[4];
b1 = w[5]; b2 = w[6]; o1 = w[7]; o2 = w[8]; ob = w[9]


# Loop over the number of epochs (how many passes over training
    data)
for ( k in 1:epochs ){

# Loop over input/training data set
    for ( i in 1:4 ){
        yd = predict (x1[i], x2[i])
        train ( x1[i], x2[i], ymodel[i], delta )

    }

# Print diagnostic output if needed:
    if ( k%%errorprintout == 0 ) { # Note: %% is the "mod"
       operator in R
        mse = 0
        for ( i in 1:4 ) {
            yhat = predict (x1[i], x2[i])
            mse = mse + (ymodel[i] - yhat)^2
        }
        cat("\nk=", k, ", error=", mse/4)
    }

# Exit epochs loop if error tolerance is met:
    if (mse/4 <= errortol) {break}
}

# Print out results:
cat("\n\nThe␣user␣parameters␣are:␣delta=", delta,
    ",␣epochs=", epochs, ",␣errortol=", errortol)

cat("\n\nCheck␣the␣neural␣network's␣predictions:")
for ( i in 1:4 ){
    yout = predict (x1[i], x2[i])
    cat("\n(x1,␣x2)=", "(", x1[i], x2[i], ")", "==>␣y␣=", round
       (yout))
}

cat("\n\nThe␣optimized␣weights␣are:")
cat("\nw11=",w11, ",␣w12=", w12, "w21=", w21, "w22=", w22,
    "\nb1=", b1, "b2=", b2, "o1=", o1, "o2=", o2, "ob=", ob)
```

1.0.3 Beyond the basics

Our discussions thus far have provided just a bare minimun glimpse into the world of neural nets. However, they provide a sufficient foundation on which we can build and explore more advanced methods. There are several different directions in which we could go from here. In this section we will introduce some of these and offer pointers for further study.

Alternate cost functions

The back propagation learning algorithm discussed in the previous sections works by minimizing a cost function. In our examples, the mean square error was used as the cost function. While this is a reasonable choice, there are situations in which it contributes to slow convergence behavior. An excellent exposition on how and why this happens is given in reference [5]. A common situation in which the learning algorithm slows down is when the computed solution is far from convergence, but the cost function still produces very small gradients. Recall, the magnitude of the gradients also determines the magnitude of the solution updates. When the iterative approximation is far from the true solution, smaller gradients translate to smaller updates and slower convergence.

An alternative cost function that is often used in conjunction with the sigmoid activation function is

$$e(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln \hat{y}_i + (1-y_i)\ln(1-\hat{y}_i)\right] \tag{1.12}$$

where the summation is carried out over n training input sets, y_i is the target output for the

i^{th} input, and $\hat{y}_i$ is the predicted output. This is known as the cross-entropy cost function. For comparison, we had defined the *MSE* cost function as

$$e(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1.13}$$

It turns out that the cross-entropy cost function often converges faster, especially in cases where the *MSE* cost function tends to stagnate. The reason, essentially, comes down to comparing the gradients produced by each cost function in these situations of interest. We elucidate some of these aspects in the exercises, but omit the details, which the interested reader can find in the references.

Overfitting

Neural networks that contain many independent parameters (weights and biases) have a tendency to exhibit a problem known as overfitting. One easy way to understand what this problem is, and why it occurs, is to consider modeling an x-y relationship such as the one seen in Figure 1.4. Shown below is a copy of a similar plot, together with one containing a random subset of 8 points from that same plot. Suppose we use this subset of 8 points for

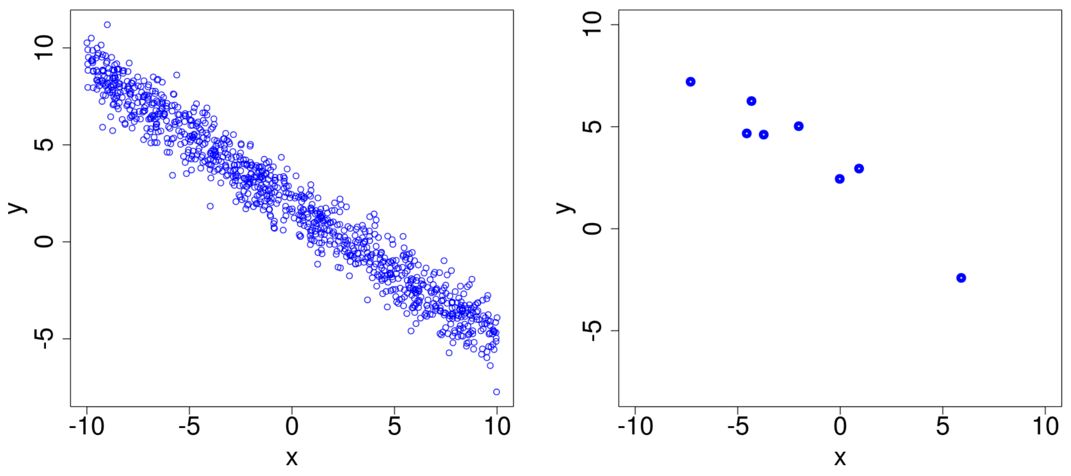


Figure 1.12: Example of an x-y relationship: Left plot shows the full dataset, and right plot shows a subset of 8 points from the same set.

training a neural network to model the relationship shown on the left. If there are enough free parameters, it is possible to fit the training data exactly, with 0 error. For instance, a polynomial of degree 7 or higher could be found such that it passes through every point in the training data set (see Figure 1.13). However, such a polynomial would do a poor job modeling the underlying relationship of interest, which would be much better approximated by a linear model. Similarly, if we train a large neural network that contains many weights and biases, it risks producing a model that does a great job fitting the training data set, but

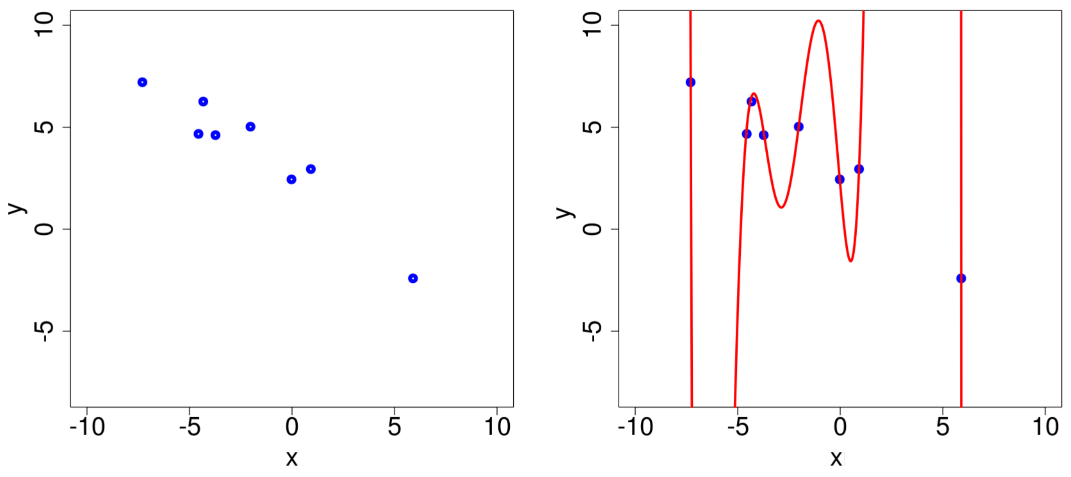


Figure 1.13: An exact polynomial fit for the subset in Figure 1.12 (shown on the left).

fails to capture essential features of the underlying application. This is a classic example of overfitting the network to the training data, and it shows how this can result in a poor model. Large neural networks often contain many thousands (if not millions!) of weights and biases, and it is not hard to see why they are susceptible to the risk of overfitting. As the power of computing technology continues to grow, it becomes possible to train extremely large networks to address more and more complex real-world applications. Thus, it is imperative to use some form of effective strategy to monitor and control overfitting, and to increase the model reliability.

Common strategies to detect and/or reduce overfitting include:

- The use of a test data set, separate and independent from the training data. This makes is possible to monitor the model's accuracy using one data set, while training it using the other.

- The use of larger training data sets. This is helpful because, with a given number of free parameters, it is harder to overfit larger sets.

- Another strategy that is somewhat complementary to the previous one is to reduce the size of the network. This would reduce the number of free parameters, and make it less prone to overfit a given size training dataset. A drawback of this approach is that the potential power and flexibility of the network may also decrease.

- More sophisticated strategies include the use of regularization techniques, as discussed in the next section.

Regularization

A primary objective of regularization methods is to mitigate the problem of overfitting, and to improve the prediction accuracy of neural networks. These techniques are designed to help even when the size of the network, and the training data set is fixed. We will introduce two of the most commonly used regularization techniques here: L1 and L2 regularization.

Recall, the training process consists of finding optimal parameter values that minimize the error, or cost function. Mathematically, this is an optimization problem with no constraints. Both L1 and L2 regularization essentially amount to imposing a constraint on the optimization that keeps the magnitude of the weights small. Why is that helpful, you might ask. One reason is that we prefer models whose output doesn't change in dramatic ways when there are small changes in the input. If we keep the magnitude of the weights small, then the output will respond in more predictable ways for incremental changes in input.

The usual strategy for implementing L2 regularization is to augment the cost function with an additional term, often known as a penalty term in optimization jargon. For example, if we use the cross-entropy cost function, here is what its regularized form would look like

$$C(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln \hat{y}_i + (1-y_i)\ln(1-\hat{y}_i)\right] \; + \; \frac{\lambda}{2n}\sum_{w} w^2 \tag{1.14}$$

Comparing this with equation 1.12, we have added a penalty term that consists of a constant times the sum of the squares of all the weights in the network. Here $\lambda > 0$ is known as the regularization parameter, and n denotes the number of training input sets, as before. Note that the penalty term includes only the *weights*, and not the *biases* in the network.

In general, any other choice of cost function can also be regularized using the same form of the penalty term. A question that naturally arises here is, how do we choose the value of λ? It is helpful to first look at what role λ is playing at the big-picture level. We may view the regularized cost function as consisting of two strictly positive terms

$$C(\mathbf{w}) = E \; + \; \lambda P$$

where E is the original cost function and P is the penalty contribution. When we minimize $C(\mathbf{w})$, ideally we want to reduce both E and P. The value of λ determines the relative importance of the two terms. When λ is small, the optimization algorithm places more emphasis on reducing E than on keeping the weights small. Conversely, when λ is large, the emphasis is reversed. That understanding can help guide the choice of λ, though it still doesn't offer a strategy to estimate numerical values for it. Typically this is done empirically, as it would depend on the choice of E (*MSE*, cross-entropy, etc.), the size of the training data set, the risks associated with overfitting, and other factors.

L1 regularization works in a very similar way, the only difference being in the precise form of the penalty term. For the cross-entropy cost function, the L1 regularized form is typically taken to be

$$C(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \ln \hat{y}_i + (1-y_i)\ln(1-\hat{y}_i)\right] \;+\; \frac{\lambda}{n}\sum_{w}|w| \tag{1.15}$$

where the penalty term contains the sum of the absolute value of all the weights. As before, $\lambda > 0$ is the regularization parameter, and it plays a similar role in mediating the tradeoff between the two terms. Its numerical value is chosen using strategies similar to those of L2 regularization.

There are other methods of regularization that are based on very different ideas than modifying the cost function. The topic of regularization remains an active area of research, both for the purpose of developing new methods, and from the standpoint of answering many open questions about existing methods. But, it seems clear from the practical experience of using regularization that these techniques can be enormously helpful for increasing the accuracy and reliability of neural network models. For more in depth study, we recommend references [2, 3, 5].

Better initial guesses

The choice of initial guess for the weights used in learning algorithms often does have an impact on convergence behavior. It is common practice to use some component of randomization for this, as was done in some of our examples. But, even with randomization, there are several choices to be made: What range of numbers to use? What type of distribution? Should the weights and biases be treated the same way?

In our examples, we typically used uniformly distributed random numbers in the range $(-1, 1)$ or $(-0.5, 0.5)$ for both weights and biases. This worked well enough for our purpose, but would it be effective for large-scale problems containing thousands of weights and biases? In general, it is common practice to use small random numbers to initialize the weights. The initial values of the biases turns out to be less important, and they are often just set to 0, or treated the same way as the weights. Recall, we also mentioned earlier that initializing the weights to 0, or to any constant value, is generally not a good idea. This is because if all the neurons within a given layer have the same set of weights and biases, and they receive the same inputs, the learning algorithm will produce the same updates for all the weights.

One common strategy for random initialization of weights is to use the standard normal distribution (with mean 0 and standard deviation 1). This is a reasonable approach to adopt as a default option. Several other variations of random initialization have been proposed in the literature, often dependent on the choice of activation function. More details can be found in the references.

Other optimization strategies

As we have seen, the heart of neural network modeling consists of the learning algorithm that finds optimal values for the network parameters. In practice, this is a large-scale optimization problem, for which many techniques have been developed. Among others, this includes several variants of gradient descent, such as stochastic gradient descent, mini-batch stochastic gradient descent, momentum based acceleration of gradient descent, and more. In this section we will take a brief look at some of these alternatives, primarily from the standpoint of a big-picture view.

A key objective in the quest for new optimizers is to speed up the convergence rate. The classic gradient descent method tends to converge prohibitively slowly when the problem size gets very large. To see why, let us consider the *MSE* cost function again

$$e(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where $\mathbf{w}$ is now a large vector, and we are summing the output error $(y_i - \hat{y}_i)^2$ over n training input sets. To complete one iteration of the optimization process and update the weights we must compute

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \frac{\delta}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)\boldsymbol{\nabla}\hat{y}_i$$

Observe that $\boldsymbol{\nabla}\hat{y}_i$ is a large vector of the same size as $\mathbf{w}$. If n is also large, as we would prefer for higher reliability, then a single iteration requires n computations of $\boldsymbol{\nabla}\hat{y}_i$, each component of which may involve several weights and biases. At the end of this we get one update for $\mathbf{w}$. Then we must repeat the process, usually for many thousands of iterations.

Stochastic and mini-batch stochastic gradient descent try to reduce the computational effort by averaging the gradient over fewer than n inputs. Typically a random subset of m inputs (with $m \ll n$) is selected and the gradient is estimated by averaging over that subset or "mini-batch." The weights are then updated, followed by selecting another random mini-batch to repeat the process. There are many variations on precisely how this is done, and how it is implemented. We leave the details to the references (see, for example, [3] and [5]).

Several other optimization methods exist, some of which are modifications or acceleration techniques for gradient descent, and others that approach the problem in completely different ways. In particular, as problem sizes get larger, there is growing interest in gradient-free methods, many of which have been in existence in the general optimization literature for decades. Due to the critical role of optimizers in training neural networks, this is an active area of research, with new developments frequently emerging.

Open-source software frameworks

The computer programming side of artificial neural networks has become a vast field of study in itself, almost independent of the mathematical and statistical methods underlying these models. While Python remains the most popular choice of programming language, writing code from scratch is not the easiest or most efficient way to explore and develop advanced neural network models. For this reason, quite a few high quality, reputable open-source software frameworks have been developed for the purpose of designing, exploring and implementing ANN models. They offer the ability to custom-build networks with any structure, and with a wide range of choice of activation functions, initialization methods, regularization, optimizers, and more.

Two of the most widely used software packages of this type are `TensorFlow` and `PyTorch`, both of which consist of a collection of libraries implemented in Python. `TensorFlow` was originally developed by Google for internal use, and later turned into an open-source product available through the TensorFlow repository in `Github`. `PyTorch` developed in a more hybrid way, with some roots in the artificial intelligence research division of Facebook/Meta Platforms.

Learning to use these packages and their various features is largely a technical endeavor, and beyond the scope of this book. In fact, due to the high popularity and large user community of both `TensorFlow` and `PyTorch`, there is an extensive collection of free, high quality learning resources available online. This includes demos, tutorials, training modules, short courses, and more. For a good starting point, we recommend the resources at TensorFlow.org and PyTorch.org. The Neural Network Playground at TensorFlow.org offers a particularly easy and interesting way to explore and interactively build multilayer networks for a collection of examples.

Exercises

1. Construct a 1-neuron network that models the logical connective AND with two inputs. Include a truth table and schematic sketch of your network showing the inputs, outputs, and numerical values of your weights and biases. Show that your network correctly models the AND connective for all possible input pairs.

2. In the previous exercise you had to choose a suitable activation function, and deduce the values of the weights and biases. Suppose, instead, we want to find the values by training the network using gradient descent. Assume the activation function is $f(x) = \tanh(x)$.

   (a) Let (x_1, x_2) denote an input pair, and $\hat{y}$ the predicted output. Introduce appropriate notation for the weights and biases, and sketch the network. See Figure 1.7 for an analogous example for the OR connective.

   (b) Suppose the cost function is $e(\mathbf{w}) = \frac{1}{2}(y - \hat{y})^2$, where $\mathbf{w}$ denotes the vector of weights and biases, y is the exact/theoretical output, and $\hat{y}$ the predicted output. Find an expression for $\boldsymbol{\nabla} e$ in terms of the weights, biases and inputs shown in your sketch.

   (c) Write an expression for updating the weight vector $\mathbf{w}^{(k)}$ at the k^{th} iteration of gradient descent.

Answers: (a) The sketch in Figure 1.7 would work here as well.
(b) Let $\mathbf{x} = (x_1, x_2, x_3)^T$, $\mathbf{w} = (w_1, w_2, w_3)^T$. Then $\hat{y} = \tanh(\mathbf{x} \cdot \mathbf{w})$, and

$$\boldsymbol{\nabla} e = -(y - \hat{y})\ \boldsymbol{\nabla} \hat{y} = -(y - \hat{y})\ [1 - \tanh^2(\mathbf{x} \cdot \mathbf{w})]\ \mathbf{x}$$

Note that $-(y - \hat{y})\ [1 - \tanh^2(\mathbf{x} \cdot \mathbf{w})]$ is a scalar quantity, and $\mathbf{x}$ is a vector of size $\boldsymbol{\nabla} e$.
(c) $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \delta\ \boldsymbol{\nabla} e$
where δ is the learning rate parameter, and $\boldsymbol{\nabla} e$ is evaluated at $\mathbf{w}^{(k)}$.

3. Suppose we wanted to extend the “and” connective to 3 inputs, say x_1 and x_2 and x_3. Describe a neural network strategy for modeling this.

4. The purpose of this exercise is to get a better understanding of why the cross-entropy cost function works as a reasonable measure of prediction error. Consider training a neural network for a classification type of application (with binary output of 0 or 1). Let y_i and $\hat{y}_i$ respectively denote the actual and predicted output for the i^{th} input, so that the cross-entropy function with n input sets has the form given in equation 1.12. Assume the standard sigmoid activation function is used for computing $\hat{y}_i$.

   (a) What are all the possible values of y_i? What about $\hat{y}_i$?

   (b) Consider a situation where $\hat{y}_i$ is a very good prediction for y_i:
   Give an example of y_i and $\hat{y}_i$ values for such a situation.
   Plug your example values into the cross-entropy expression and find its numerical value. Let $n = 1$ for this purpose.

   (c) Consider the opposite situation, so that $\hat{y}_i$ is a very poor prediction for y_i:
   Give an example of y_i and $\hat{y}_i$ values for such a situation.
   Plug your example values into the cross-entropy expression and find its numerical value.

   (d) Write a couple of sentences summarizing how the numerical value of the cross-entropy is related to the accuracy of predictions.

Answers: (a) $y_i = \{0, 1\}$, $\hat{y}_i = (0, 1)$. Note that y_i is binary, whereas $\hat{y}_i$ is continuous.
(b) There are infinite possible answers. Here are some: $(y_i, \hat{y}_i) = (1, 0.981)$, OR $(y_i, \hat{y}_i) = (1, 0.97)$, OR $(y_i, \hat{y}_i) = (0, 0.019)$. If we pick $(y_i, \hat{y}_i) = (1, 0.981)$, then

$$e = -\left[y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)\right] = -\left[1 \cdot \ln(0.981) + (1 - 1) \ln(0.981)\right] \approx 0.019$$

(c) Again, there are infinite possible answers: $(y_i, \hat{y}_i) = (1, 0.09)$, $(y_i, \hat{y}_i) = (1, 0.003)$, $(y_i, \hat{y}_i) = (0, 0.98)$. If we pick $(y_i, \hat{y}_i) = (1, 0.09)$, then

$$e = -\left[1 \cdot \ln(0.09) + (1 - 1) \ln(0.09)\right] \approx 2.408$$

(d) When the predicted value is close to the actual value, the cross-entropy function has values close to 0. Conversely, when the actual and predicted values are not close, the cross-entropy function has large values.

5. One reason why the cross-entropy cost function works better than the *MSE* is that it typically produces a large correction to the weights when the prediction error is large. To see this, let us compare the gradients produced by the two functions in a situation with large prediction error.

   Consider training a 1-neuron network for a classification problem with binary output of 0 or 1. Let y_i and $\hat{y}_i$ respectively denote the actual and predicted output in the back propagation algorithm for the i^{th} input. [**incomplete**]

6. Find an expression for the derivatives $\frac{\partial e}{\partial y_1}$ and $\frac{\partial e}{\partial y_2}$ for each of the following

   (a) $e = \sigma(z), \quad z = -0.3y_1 + 0.8y_2 - 2.5$. Here σ represents the sigmoid function.

   (b) $e = \tanh(z), \quad z = 1.2u + 3v - 27, \quad u = \tanh(y_1), \quad v = \tanh(y_2)$

   (c) $e = \sigma(z), \quad z = -0.97u - 3.5v + 7, \quad u = \sigma(w), \quad v = \sigma(x),$
   $w = 2y_1 + 5, \quad x = -1.2y_2 - 3.$

   (d) $e = -\dfrac{1}{2}(6 - \hat{y})^2, \quad \hat{y} = \tanh(z), \quad z = 2u - v - 1, \quad u = -y_2/4, \quad v = (y_1 - 5)/2$

   Answers:

   (a) $\frac{\partial e}{\partial y_1} = -0.3\,\sigma(z)(1-\sigma(z)); \;\; \frac{\partial e}{\partial y_2} = 0.8\,\sigma(z)(1-\sigma(z))$
   where $z = -0.3y_1 + 0.8y_2 - 2.5$ and $\sigma(z) = \frac{1}{1+e^{-z}}$

   (b) $\frac{\partial e}{\partial y_1} = 1.2\,(1-\tanh^2 z)\,(1-\tanh^2 y_1); \;\; \frac{\partial e}{\partial y_2} = 3\,(1-\tanh^2 z)\,(1-\tanh^2 y_2)$
   where $z = 1.2u + 3v - 27$, with $u = \tanh(y_1)$, $v = \tanh(y_2)$

   (c) $$\begin{aligned}\frac{\partial e}{\partial y_1} &= \sigma(z)(1-\sigma(z))\cdot(-0.97)\cdot\sigma(w)(1-\sigma(w))\cdot(2)\\ &= -1.94\,\sigma(z)\sigma(w)(1-\sigma(z))(1-\sigma(w)),\\ \frac{\partial e}{\partial y_2} &= \sigma(z)(1-\sigma(z))\cdot(-3.5)\cdot\sigma(x)(1-\sigma(x))\cdot(-1.2)\\ &= 4.2\,\sigma(z)\sigma(x)(1-\sigma(z))(1-\sigma(x)),\end{aligned}$$
   with z, w, x as given in the problem.

   (d) $$\frac{\partial e}{\partial y_1} = (6-\hat{y})\cdot(1-\tanh^2 z)\cdot(-1)\cdot(1/2) \;=\; -\frac{(6-\hat{y})\,(1-\tanh^2 z)}{2}$$
   $$\frac{\partial e}{\partial y_2} = (6-\hat{y})\cdot(1-\tanh^2 z)\cdot(2)\cdot(-1/4) \;=\; -\frac{(6-\hat{y})\,(1-\tanh^2 z)}{2}$$
   with $\hat{y}, z$ as given in the problem.

7. Consider the neural network shown below, in which a single input value, say x, is transmitted by the input neuron to the next layer (see the example in Figure 1.9 for a very similar situation). The sketch shows the values of all the weights and biases in the network. Following standard convention, each bias value of 1 is multiplied by the corresponding weight shown in the sketch to obtain its input value. For example, the net input that neuron h_1 receives is $-3x + 1.5$. Neurons h_1, h_2 and O include an activation function that is not shown in the sketch. Compute the output $\hat{y}$ when $x = 2$ for each of the following choices of activation function (show calculation details at each neuron):

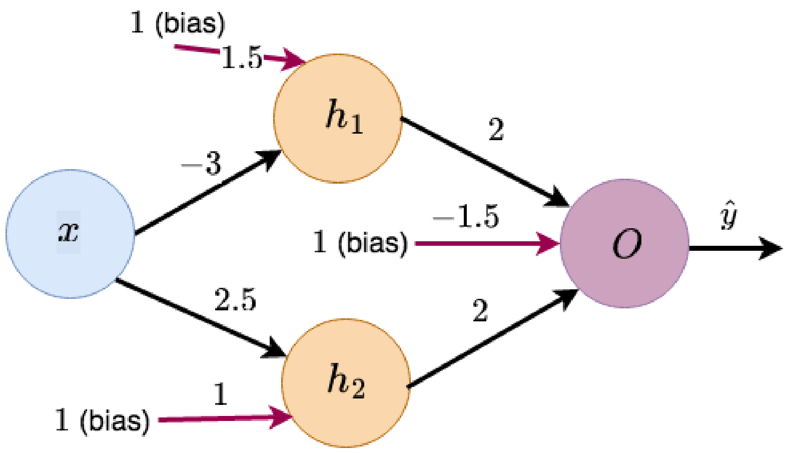


Figure 1.14: Neural network exercise.

(a) Threshold function: $f(x) = \begin{cases} 0, & \text{if } x < 0 \\ 1, & \text{if } x \geq 0 \end{cases}$

(b) ReLU: $f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{cases}$

(c) Sigmoid: $f(x) = \sigma(x) = \dfrac{1}{1 + e^{-x}}$

8. This is an extension of the previous exercise in which we will carry out one iteration of gradient descent optimization as part of training the network. Suppose the initial values of the weights and biases is as shown in Figure 1.9. Assume a sigmoid activation function, and suppose a training data set inputs the value $x_i = 2$, with target output value $y_i = 1$.

   (a) Find the predicted output $\hat{y}_i$. (Hint: See your answers to the previous exercise!)

   (b) Assume the cost function is $e(\mathbf{w}) = -\frac{1}{2}(y_i - \hat{y}_i)^2$, where $\mathbf{w}$ denotes the vector of weights and biases. Compute the numerical value of $\boldsymbol{\nabla} e$.

   (c) Compute the updated values of the weights and biases using one step of gradient descent. Pick some reasonable value for the learning rate parameter δ.

9. Many important real-world applications can be framed as classification problems, wherein we want to determine the type of output for some given set of inputs. **[incomplete]**

(The exercises section is not yet complete.)

Bibliography

[1] J. Brownlee, “Statistical Methods for Machine Learning,” Machine Learning Mastery, 2019.

[2] J. Dawani, “Hands-On Mathematics for Deep Learning,” Packt Publishing, 2020.

[3] I. Goodfellow, Y. Bengio and A. Courville, “Deep Learning,” MIT Press, 2016.

[4] M. Deisenroth, A. Faisal and C.S. Ong, “Mathematics for Machine Learning,” Cambridge University Press, 2020.

[5] M.A. Nielsen, “Neural Networks and Deep Learning,” Determination Press, 2015.