**Artificial Neural Networks**

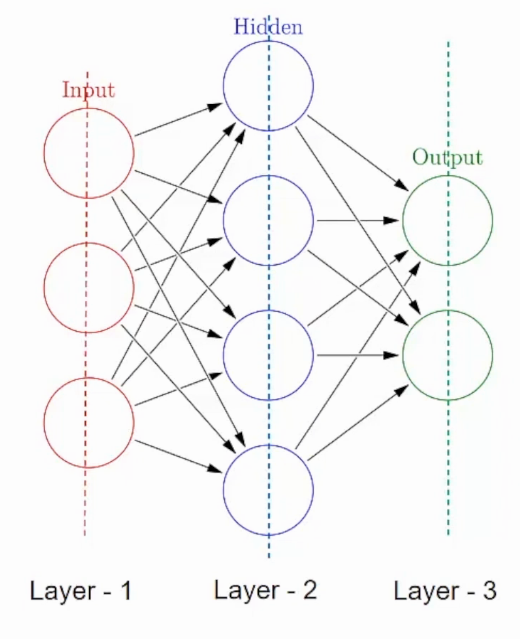
**The Input Layer and Dense Layers 99 A**

## Building Blocks of a Neural Network: Layers and Neurons

There are two building blocks of a Neural Network, let’s look at each one of them in detail-

### 1. What are Layers in a Neural Network?

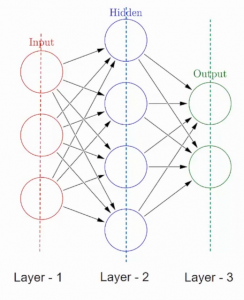
A neural network is made up of vertically stacked components called **Layers**. Each dotted line in the image represents a layer. There are three types of layers in a NN-



**Input Layer**– First is the input layer. This layer will accept the data and pass it to the rest of the network.

**Hidden Layer**– The second type of layer is called the hidden layer. Hidden layers are either one or more in number for a neural network. In the above case, the number is 1. Hidden layers are the ones that are actually responsible for the excellent performance and complexity of neural networks. They perform multiple functions at the same time such as data transformation, automatic feature creation, etc.

**Output layer**– The last type of layer is the output layer. The output layer holds the result or the output of the problem. Raw images get passed to the input layer and we receive output in the output layer. For example-

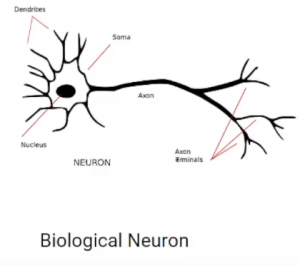


In this case, we are providing an image of a vehicle and this output layer will provide an output whether it is an emergency or non-emergency vehicle, after passing through the input and hidden layers of course.

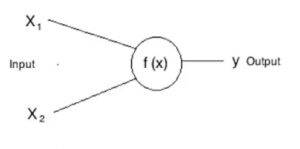
Now, that we know about layers and their function let’s talk in detail about what each of these layers is made up of.

### 2. What are Neurons in a Neural Network?

A layer consists of small individual units called neurons.A **neuron** in a neural network can be better understood with the help of biological neurons. An artificial neuron is similar to a biological neuron. It receives input from the other neurons, performs some processing, and produces an output.



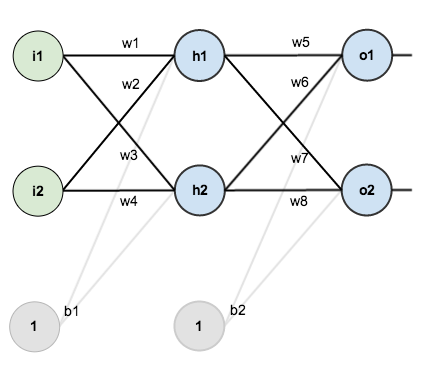
Now let’s see an artificial neuron-



Here, **X1** and **X2** are inputs to the artificial neurons, **f(X)** represents the processing done on the inputs and **y**represents the output of the neuron.

A neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.

Here’s the basic structure:



In order to have some numbers to work with, here are the initial weights, the biases, and training inputs/outputs:



The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

For the rest of this tutorial we’re going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

## The Forward Pass

To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we’ll feed those inputs forward though the network.

We figure out the total net input to each hidden layer neuron, squash the total net input using an activation function (here we use the logistic function), then repeat the process with the output layer neurons.

Here’s how we calculate the total net input for h_1:

net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1

net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775

We then squash it using the logistic function to get the output of h_1:

out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992

Carrying out the same process for h_2 we get:

out_{h2} = 0.596884378

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Here’s the output for o_1:

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967

out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507

And carrying out the same process for o_2 we get:

out_{o2} = 0.772928465

### Calculating the Total Error

We can now calculate the error for each output neuron using the [squared error function](http://en.wikipedia.org/wiki/Backpropagation#Derivation) and sum them to get the total error:

E_{total} = \sum \frac{1}{2}(target - output)^{2}

[Some sources](http://www.amazon.com/Introduction-Math-Neural-Networks-Heaton-ebook/dp/B00845UQL6/ref=sr_1_1?ie=UTF8&qid=1426296804&sr=8-1&keywords=neural+network) refer to the target as the *ideal* and the output as the *actual*.

The \frac{1}{2} is included so that exponent is cancelled when we differentiate later on. The result is eventually multiplied by a learning rate anyway so it doesn’t matter that we introduce a constant here [[1](http://en.wikipedia.org/wiki/Backpropagation#Derivation)].

For example, the target output for o_1 is 0.01 but the neural network output 0.75136507, therefore its error is:

E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^{2} = \frac{1}{2}(0.01 - 0.75136507)^{2} = 0.274811083

Repeating this process for o_2 (remembering that the target is 0.99) we get:

E_{o2} = 0.023560026

The total error for the neural network is the sum of these errors:

E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109

## The Backwards Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

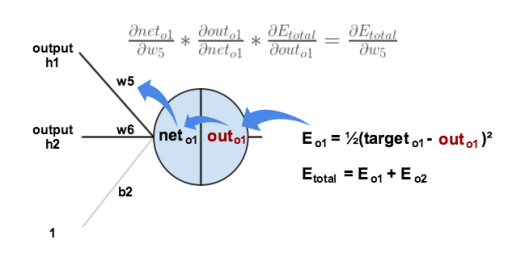
### Output Layer

Consider w_5. We want to know how much a change in w_5 affects the total error, aka \frac{\partial E_{total}}{\partial w_{5}}.

\frac{\partial E_{total}}{\partial w_{5}}is read as “the partial derivative of E_{total} with respect to w_{5}“. You can also say “the gradient with respect to w_{5}“.

By applying the [chain rule](http://en.wikipedia.org/wiki/Chain_rule) we know that:

Visually, here’s what we’re doing:



We need to figure out each piece in this equation.

First, how much does the total error change with respect to the output?

E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^{2} + \frac{1}{2}(target_{o2} - out_{o2})^{2}

\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2 - 1} * -1 + 0

\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507

-(target - out)is sometimes expressed as out - target

When we take the partial derivative of the total error with respect to out_{o1}, the quantity \frac{1}{2}(target_{o2} - out_{o2})^{2} becomes zero because out_{o1} does not affect it which means we’re taking the derivative of a constant which is zero.

Next, how much does the output of o_1 change with respect to its total net input?

The partial [derivative of the logistic function](http://en.wikipedia.org/wiki/Logistic_function#Derivative) is the output multiplied by 1 minus the output:

out_{o1} = \frac{1}{1+e^{-net_{o1}}}

\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602

Finally, how much does the total net input of o1 change with respect to w_5?

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

\frac{\partial net_{o1}}{\partial w_{5}} = 1 * out_{h1} * w_5^{(1 - 1)} + 0 + 0 = out_{h1} = 0.593269992

Putting it all together:

\frac{\partial E_{total}}{\partial w_{5}} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041

You’ll often see this calculation combined in the form of the [delta rule](http://en.wikipedia.org/wiki/Delta_rule):

\frac{\partial E_{total}}{\partial w_{5}} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}

Alternatively, we have \frac{\partial E_{total}}{\partial out_{o1}} and \frac{\partial out_{o1}}{\partial net_{o1}} which can be written as \frac{\partial E_{total}}{\partial net_{o1}}, aka \delta_{o1} (the Greek letter delta) aka the *node delta*. We can use this to rewrite the calculation above:

\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})

Therefore:

\frac{\partial E_{total}}{\partial w_{5}} = \delta_{o1} out_{h1}

Some sources extract the negative sign from \delta so it would be written as:

\frac{\partial E_{total}}{\partial w_{5}} = -\delta_{o1} out_{h1}

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we’ll set to 0.5):

w_5^{+} = w_5 - \eta * \frac{\partial E_{total}}{\partial w_{5}} = 0.4 - 0.5 * 0.082167041 = 0.35891648

[Some](http://en.wikipedia.org/wiki/Delta_rule) [sources](http://aima.cs.berkeley.edu/) use \alpha (alpha) to represent the learning rate, [others use](https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf) \eta (eta), and [others](http://web.cs.swarthmore.edu/~meeden/cs81/s10/BackPropDeriv.pdf) even use \epsilon (epsilon).

We can repeat this process to get the new weights w_6, w_7, and w_8:

w_6^{+} = 0.408666186

w_7^{+} = 0.511301270

w_8^{+} = 0.561370121

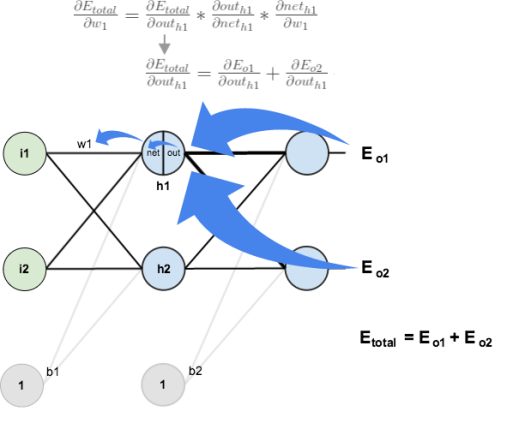
We perform the actual updates in the neural network *after* we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

### Hidden Layer

Next, we’ll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.

Big picture, here’s what we need to figure out:

Visually:

[](https://matthewmazur.files.wordpress.com/2015/03/nn-calculation.png)

We’re going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h1} affects both out_{o1} and out_{o2} therefore the \frac{\partial E_{total}}{\partial out_{h1}} needs to take into consideration its effect on the both output neurons:

Starting with \frac{\partial E_{o1}}{\partial out_{h1}}:

We can calculate \frac{\partial E_{o1}}{\partial net_{o1}} using values we calculated earlier:

And \frac{\partial net_{o1}}{\partial out_{h1}} is equal to w_5:

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40

Plugging them in:

Following the same process for \frac{\partial E_{o2}}{\partial out_{h1}}, we get:

\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119

Therefore:

Now that we have \frac{\partial E_{total}}{\partial out_{h1}}, we need to figure out \frac{\partial out_{h1}}{\partial net_{h1}} and then \frac{\partial net_{h1}}{\partial w} for each weight:

out_{h1} = \frac{1}{1+e^{-net_{h1}}}

\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999 ) = 0.241300709

We calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron:

net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1

\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05

Putting it all together:

\frac{\partial E_{total}}{\partial w_{1}} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568

You might also see this written as:

\frac{\partial E_{total}}{\partial w_{1}} = \delta_{h1}i_{1}

We can now update w_1:

w_1^{+} = w_1 - \eta * \frac{\partial E_{total}}{\partial w_{1}} = 0.15 - 0.5 * 0.000438568 = 0.149780716

Repeating this for w_2, w_3, and w_4

w_2^{+} = 0.19956143

w_3^{+} = 0.24975114

w_4^{+} = 0.29950229

Finally, we’ve updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

**Hot Dog-Detecting Dense Network 101**

**Forward Propagation Through the First Hidden Layer 102**

**Forward Propagation Through Subsequent Layers 103**

**The Softmax Layer of a Fast Food-Classifying Network 106**

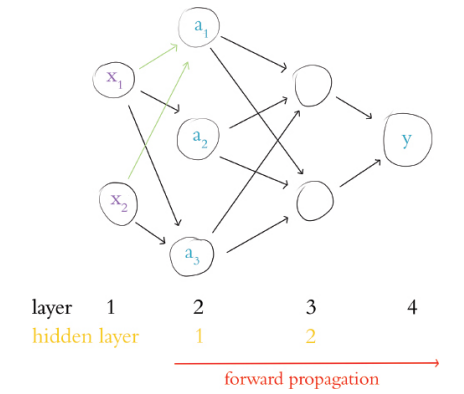
A frivolous hot dog­detecting binary classifier and the mathematical notation we used to define artificial neurons. As shown in Figure 7.1, our hot dog classifier is no longer a single neuron; in this chapter, it is a dense network of artificial neurons. More specifically, with this network architecture: œ We have reduced the number of input neurons down to two for simplicity: œ The first input neuron, x , represents the volume of ketchup (in, say, milliliters, which abbreviates to mL) on the object being considered by the network. (We are no longer working with perceptrons, so we are no longer restricted to binary inputs only.)

œ The second input neuron, x , represents mL of mustard. œ We have two dense hidden layers:

œ The first hidden layer has three ReLU neurons.

œ The second hidden layer has two ReLU neurons.

œ The output neuron is denoted by ŷ in the network. This is a binary classification problem, so—as outlined in the previous section—this neuron should be sigmoid. As in our perceptron examples in Chapter 6, y = 1 corresponds to the presence of a hot dog and y = 0 corresponds to the presence of some other object.



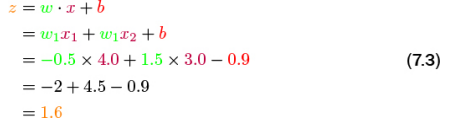
Forward Propagation through the First Hidden Layer Having described the architecture of our hot dog­detecting network, let’s turn our attention to its functionality by focusing on the neuron labelled a . This particular neuron, like its siblings a and a , receives input regarding a given object’s ketchup­yness and mustard­y­ness from x and x , respectively. Despite receiving the same data as a and a , a treats these data uniquely by having its own unique parameters. Remembering Figure 6.7, “the most important equation in this book” —w x + b—we may grasp this behavior more concretely. Breaking this equation down for the neuron labelled a , we consider that it has two inputs from the previous layer, x and x . This neuron also has two weights: w (which applies to the importance of the ketchup measurement x ) and w (which applies to the importance of the mustard measurement x ). With these five pieces of information we can calculate z, the weighted input to that neuron:



In turn, with the z value for the neuron labelled a , we can calculate the activation a it outputs. Since the neuron labelled a is a ReLU neuron, we use the equation introduced in Figure 6.11:



œ x is 4.0 mL of ketchup for a given object presented to the network œ x is 3.0 mL of mustard for that same object œ w = −0.5 œ w = 1.5 œ b = −0.9 To calculate z let’s start with Equation 7.1 and then fill in our contrived values:



Finally, to compute a—the activation output of the neuron labelled a —we can leverage Equation 7.2:



As suggested by the right­facing arrow along the bottom of Figure 7.1, executing the calculations through an artificial neural network from the input layer (the x values) through to the output layer (ŷ) is called forward propagation. Immediately above, we detailed the process for forward propagating through a single neuron in the first hidden layer of our hot dog­detecting network. To forward propagate through the remaining neurons of the first hidden layer—that is, to calculate the a values for the neurons labelled a and a —we would follow the same process as we did for the neuron labelled a . The inputs x and x are identical for all three neurons, but despite being fed the same measurements of ketchup and mustard, each neuron in the first hidden layer will output a different activation a because the parameters w , w and b vary for each of the

neurons in the layer.

**Forward Propagation through Subsequent Layers**

The process of forward propagating through the remaining layers of the network is essentially the same as propagating through the first hidden layer, but for clarity’s sake, let’s work through it together. In Figure 7.2, we’ll assume that we’ve already calculated the activation value a for each of the neurons in the first hidden layer. Returning our focus to the neuron labelled a , the activation it outputs (a1 = 1.6) becomes one of the three inputs into the neuron labelled a4 (and, as highlighted in the figure, this same activation of a = 1.6 is also fed as one of the three inputs into the neuron labelled a 5).

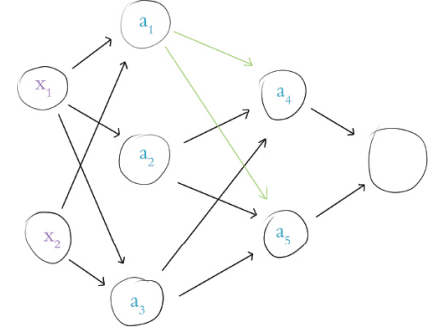
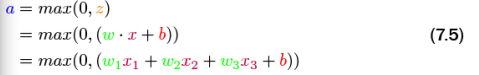


Figure 7.2 Our hot dog­detecting network from Figure 7.1, now highlighting the activation output of neuron a , which is provided as an input to both neuron a4 and neuron a5 .

To provide an example of forward propagation through the second hidden layer, let’s compute a for the neuron labelled a 4.

Again, we employ the all ­important equation w · x + b. For brevity’s sake, we’ve combined it with the ReLU activation function:



This is sufficiently similar to Equations 7.3 and 7.4 that it would be superfluous to walk through the arithmetic again with feigned values. The only twist, as we propagate through the second hidden layer, is that the layer’s inputs (i.e., x in the equation w x + b) come not from outside the network—instead they are provided by the first hidden layer. Thus, in Equation 7.5:

œ x1 is the value a = 1.6, which we obtained earlier from the neuron labelled a1

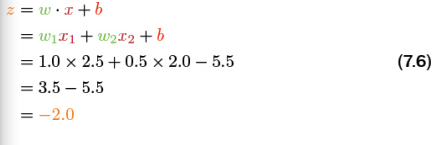
œ x2 is the activation output a (whatever it happens to equal) from the neuron labelled a2 , and

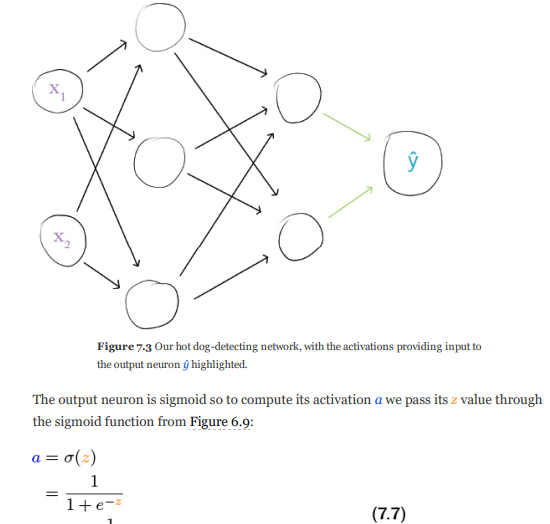
œ x3 is likewise a unique activation a from the neuron labelled a3

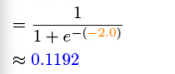
In this manner, the neuron labelled a is able to non­linearly recombine the information provided by the three neurons of the first hidden layer. The neuron labelled a5 also non­linearly recombines this information, but it would do it in its own distinctive way: The unique parameters w1 , w2 , w3 and b for this neuron would lead it to output a unique a activation of its own.

Having illustrated forward propagation through all of the hidden layers of our hot dogdetecting network, let’s round the process off by propagating through the output layer. Figure 7.3 highlights that our single output neuron receives its inputs from the neurons labelled a4 and a5 .

Let’s begin by calculating z for this output neuron. The formula is identical to Equation 7.1, which we used to calculate z for the neuron labelled a , except that the (contrived, as usual) values we plug into the variables are different:







The activation a computed by the sigmoid neuron in the output layer is a very special case because it is the final output of our entire hot dog­detecting neural network. Since it’s so special, we assign it a distinctive designation: ŷ, which is pronounced “why hat”. This value ŷ is the network’s guess as to whether the object presented to it was a hot dog or not a hot dog, and we can express this in probabilistic language. Given the inputs x and x that we fed into the network—that is, 4.0 mL of ketchup and 3.0 mL of mustard—the network estimates that there is an 11.92% chance that an object with 1 2 those particular condiment measurements is a hot dog. If the object presented to the network was indeed a hot dog (y = 1) then this ŷ of 0.1192 was pretty far off the mark. On the other hand, if the object was truly not a hot dog (y = 0) then the ŷ is quite good. We’ll formalize the evaluation of ŷ in Chapter 8, but the general notion is is that the closer ŷ is to the true value y, the better.

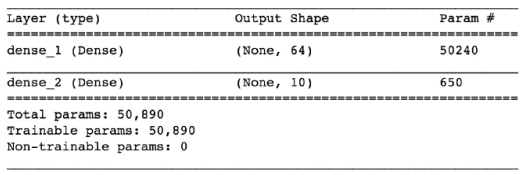
**Revisiting Our Shallow Network 108**

With the knowledge of dense networks that we developed over the course of this chapter, we can return to our Shallow Net in Keras notebook and understand the model summary within it. Example 5.2 shows the three lines of Keras code we used to architect a shallow neural network for classifying MNIST digits. As detailed in Chapter 5, over those three lines of code we instantiated a model object and added layers of artificial neurons to it. By calling the summary() on the model, we see the modelsummarizing table provided in Figure 7.5. The table has three columns:

œ Layer (type): the name and type of each of our layers

œ Output Shape: the dimensionality of the layer

œ Param #: the number of parameters (weights w and biases b) associated with the layer

****

**Figure 7.5** A summary of the model object from our “Shallow Net in Keras” Jupyter notebook. The input layer performs no calculations and never has any of its own parameters so no information on it is displayed directly. The first row in the table, therefore, corresponds 6 to the first hidden layer of the network. The table indicates that this layer:

œ is called dense\_1; this is a default name as we did not designate one explicitly

œ is a Dense layer, as we specified in Example 5.2

œ is composed of 64 neurons, as we further specified in Example 5.2

œ has 50240 parameters associated with it, broken down into:

œ 50176 weights, corresponding to each of the 64 neurons in this dense layer receiving input from each of the 784 neurons in the input layer (64\*784)

œ plus 64 biases, one for each of the neurons in the layer

œ giving us a total of 50240 nparameters = nw + nb = 50176 + 64 = 50240

The second row of the table in Figure 7.5 corresponds to the model’s output layer. The table tells us that this layer:

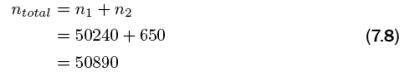
œ is called dense\_2

œ is a Dense layer,

as we specified it to be œ consists of 10 neurons—yet again, as we specified œ has 650 parameters associated with it:

œ 640 weights, corresponding to each of the ten neurons receiving input from each of the 64 neurons in the hidden layer (64\*10)

œ plus 10 biases, one for each of the output neurons From the parameter counts for each layer, we can calculate for ourselves the Total params line displayed in Figure 7.5:



All 50890 of these parameters are “Trainable params” because—during the subsequent model.fit() call in the Shallow Net in Keras notebook—they are permitted to be tuned during model training. This is the norm, but as we’ll see in Part III, there are situations where it is fruitful to freeze some of the parameters in a model rendering them “Non­trainable params”.