# 5. The (Code) Cart Ahead of the (Theory) Horse

In Part I, we provided a high-level overview of deep learning by demonstrating its use across a spectrum of cutting-edge applications. Along the way, we sprinkled in foundational deep learning concepts from its hierarchical, representation-learning nature through to its relationship to the field of artificial intelligence. Repeatedly, as we touched on a concept, we noted that in Part II of the book we would dive into the low-level theory and mathematics behind it. While we promise this *is* true, we are going to take this final opportunity to put the fun, hands-on coding cart ahead of the proverbial—in this case, theory-laden—horse.

In this chapter we do a line-by-line walk-through of a notebook of code featuring a neural network model. While you will need to bear with us because we have not yet detailed much of the theory underpinning the code, this serpentine approach will make the apprehension of theory in the subsequent chapters easier: Instead of being an abstract idea, each element of theory we introduce in this part of the book will be rooted in a tangible line of applied code.

## PREREQUISITES

Working through the examples in this book will be easiest if you are familiar with the basics of the Unix command line. These are provided by Zed Shaw in Appendix A of his deceptively enjoyable *Learn Python the Hard Way*. [1]

1 . Shaw, Z. (2013). *Learn Python the Hard Way, 3rd Ed*. New York: Addison-Wesley. This relevant appendix, Shaw's "Command Line Crash Course," is available online at learnpythonthehardway.org/book/appendixa.html (http://learnpythonthehardway.org/book/appendixa.html).

Speaking of Python, since it is comfortably the most popular software language in the data science community (at time of writing, anyway), it's the language we selected for our example code throughout the book. Python's prevalence extends across the composition of stand-alone scripts through to the deployment of machine learning models into production systems. If you're new to Python or you're feeling a tad rusty, Shaw's book serves as an appropriate general reference, while Daniel Chen's *Pandas for Everyone* [2] is ideal for learning how to apply the language to data modeling in particular.

2 . Chen, D. (2017). *Pandas for Everyone: Python Data Analysis*. New York: Addison-Wesley.

## INSTALLATION

Regardless of whether you're planning on executing our code notebooks via Unix, Linux, macOS, or Windows, we have made step-by-step installation instructions available in the GitHub repository that accompanies this book:

Click here to view code image

```
github.com/the-deep-learners/deep-learning-illustrated
```

If you'd prefer to view the completed notebooks instead of running them on your own machine, you are more than welcome to do that from the GitHub repo as well.

We elected to provide our code within the comfort of interactive Jupyter notebooks. [3] Jupyter is a common option today for writing and sharing scripts, particularly during exploratory phases in which a data scientist is experimenting with preprocessing, visualizing, and modeling her data. Our installation instructions suggest running Jupyter from within a Docker container. [4] This containerization ensures that you'll have all of the software dependencies you need to run our notebooks while simultaneously preventing these dependencies from clashing with software you already have installed on your system.

3 . jupyter.org (http://jupyter.org). We recommend familiarizing yourself with the hot keys to breeze through Jupyter notebooks with pizzazz.

4 . docker.com (http://docker.com)

## A SHALLOW NETWORK IN KERAS

To kick off the code portion of our book, we will:

1. Detail a revered dataset of handwritten digits
2. Load these data into a Jupyter notebook
3. Use Python to prepare the data for modeling
4. Write a few lines of code in Keras, a high-level deep learning API, to construct an artificial neural network (in TensorFlow, behind the scenes) that predicts what digit a given handwritten sample represents

### The MNIST Handwritten Digits

Back in Chapter 1 when we introduced the LeNet-5 machine vision architecture (Figure 1.11), we mentioned that one of the advantages Yann LeCun (Figure 1.9) and his colleagues had over previous deep learning practitioners was a superior dataset for training their model. This dataset of handwritten digits, called MNIST (see the samples in Figure 5.1), came up again in the context of being imitated by Ian Goodfellow's generative adversarial network (Figure 3.2a). The MNIST dataset is ubiquitous across deep learning tutorials, and for good reason. By modern standards, the dataset is small enough that it can be modeled rapidly, even on a laptop computer processor. In addition to their portable size, the MNIST digits are handy because they occupy a sweet spot with respect to how challenging they are to classify: The handwriting samples are sufficiently diverse and contain complex enough details that they are not *easy* for a machine-learning algorithm to identify with high accuracy, and yet by no means do they pose an insurmountable problem. However, as you will observe yourself as we make our way through Part II of this book, a well-designed deep-learning model can nearly faultlessly classify the handwriting as the appropriate digit.

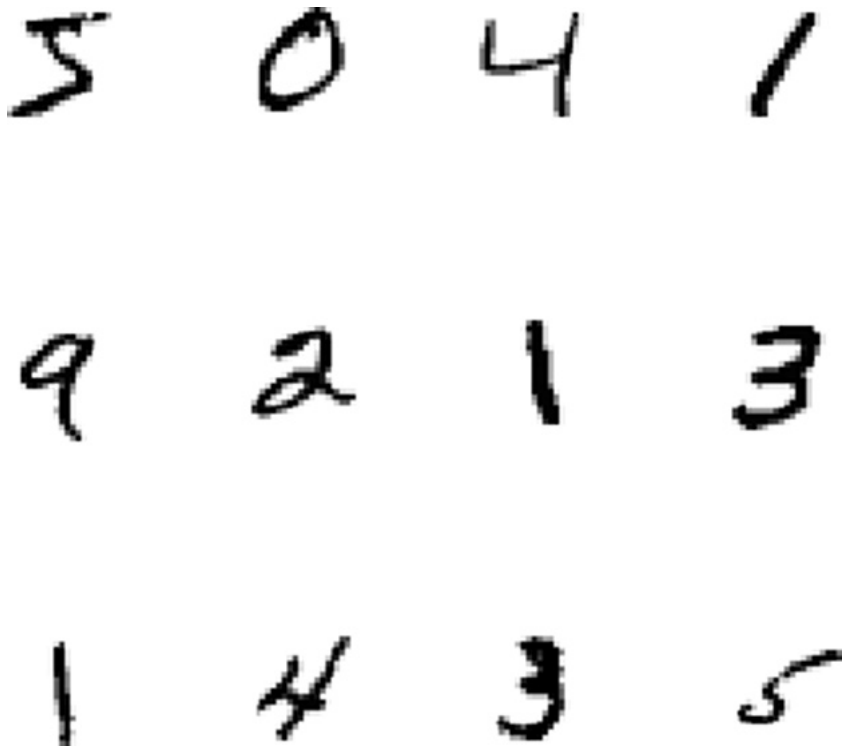

**Figure 5.1** A sample of a dozen images from the MNIST dataset. Each image contains a single digit handwritten by either a high school student or a U.S. census worker.

The MNIST dataset was curated by LeCun (Figure 1.9), Corinna Cortes (Figure 5.2), and the Microsoft-AI-researcher-turned-musician Chris Burges in the 1990s. [5] It consists of 60,000 handwritten digits for training an algorithm, and 10,000 more for validating the algorithm's performance on previously unseen data. The data are a subset (a *m*odification) of a larger body of handwriting samples collected from high school students and census workers by the U.S. National Institute of Standards and Technology (NIST).

**Figure 5.2** The Danish computer scientist Corinna Cortes is head of research at Google's New York office. Among her countless contributions to both pure and applied machine learning, Cortes (with Chris Burges and Yann LeCun) curated the widely used MNIST dataset.

5 . yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)

As exemplified by Figure 5.3, every MNIST digit is a 28×28-pixel image. [6] Each pixel is 8-bit, meaning that the pixel darkness can vary from 0 (white) to 255 (black), with the intervening range of integers representing gradually darker shades of gray.

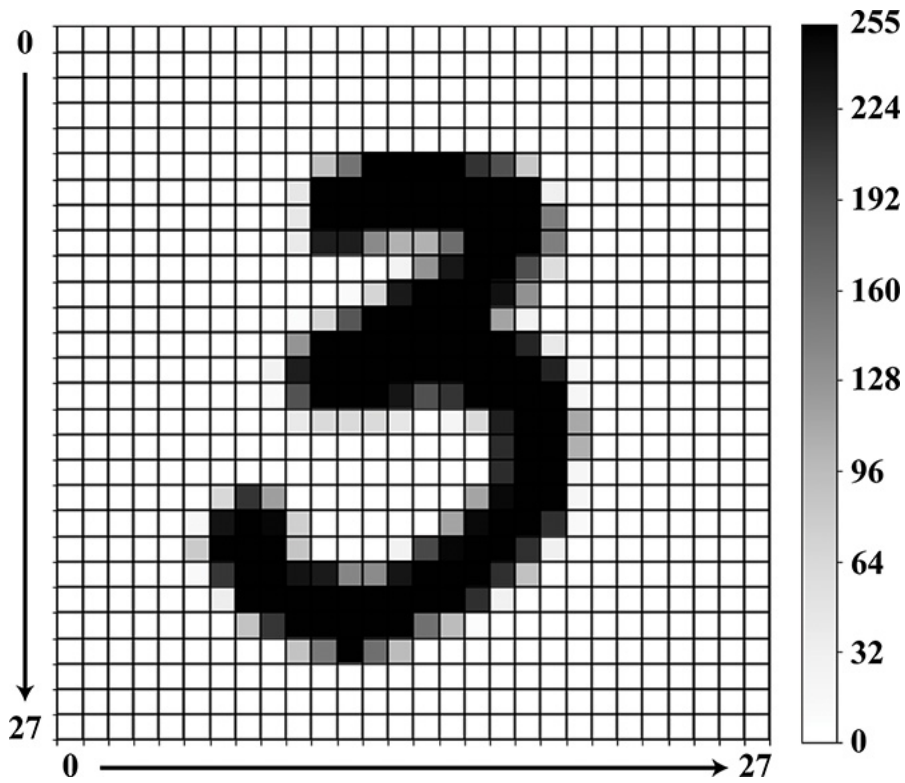


**Figure 5.3** Each handwritten MNIST digit is stored as a 28×28-pixel grayscale image. See the Jupyter notebook titled *MNIST Digit Pixel by Pixel* that accompanies this book for the code we used to create this figure.

6 . Python uses zero indexing, so the first row and column are denoted with `0`. The 28th row and 28th column of pixels are therefore both denoted with `27`.

## A Schematic Diagram of the Network

In our *Shallow Net in Keras* Jupyter notebook, [7] we create an artificial neural network to predict what digit a given handwritten MNIST image represents. As shown in the rough schematic diagram in Figure 5.4, this artificial neural network features one hidden layer of artificial neurons, for a total of three layers. Recalling Figure 4.2, with so few layers this ANN would not generally be considered a *deep* learning architecture; hence it is *shallow*.

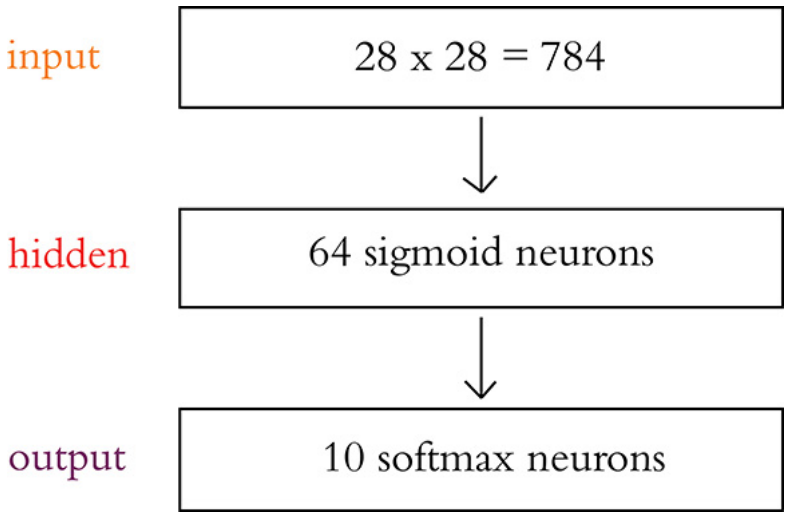


**Figure 5.4** A rough schematic of the shallow artificial-neural-network architecture we're whipping up in this chapter. We detail the particular sigmoid and softmax flavors of artificial neurons in Chapters 6 and 7 , respectively.

7 . Within this book's GitHub repository, navigate into the *notebooks* directory.

The first layer of the network is reserved for inputting our MNIST digits. Because they are 28×28-pixel images, each one has a total of 784 values. After we load in the images, we'll flatten them from their native, two-dimensional 28×28 shape to a one-dimensional array of 784 elements.

You could argue that collapsing the images from two dimensions to one will cause us to lose a lot of the meaningful structure of the handwritten digits. Well, if you argued that, you'd be right! Working with one-dimensional data, however, means we can use relatively unsophisticated neural network models, which is appropriate at this early stage in our journey. Later, in Chapter 10, you'll be in a position to appreciate more-complex models that can handle multidimensional inputs.

The pixel-data inputs will be passed through a single, hidden layer of 64 artificial neurons. [8] The number (64) and type (*sigmoid*) of these neurons aren't critical details at present; we begin to explain these model attributes in the next chapter. The key piece of information at this time is that, as we demonstrate in Chapter 1 (see Figures 1.18 and 1.19), the neurons in the hidden layer are responsible for learning representations of the input data so that the network can predict what digit a given image represents.

8 . "Hidden" layers are so called because they are not exposed; data impact them only indirectly, via the input layer or the output layer of neurons.

Finally, the information that is produced by the hidden layer will be passed to 10 neurons in the output layer. We detail how a *softmax* layer of neurons works in Chapter 7, but, in essence, we have 10 neurons because we have 10 categories of digit to classify. Each of these 10 neurons outputs a probability: one for each of the 10 possible digits that a given MNIST image could represent. As an example, a fairly well-trained network that is fed the image in Figure 5.3 might output that there is a 0.92 probability that the image is of a *three*, a 0.06 probability that it's a *two*, a 0.02 probability that it's an *eight*, and a probability of 0 for the other seven digits.

## Loading the Data

At the top of the notebook we import our software dependencies, which is the unexciting but necessary step shown in Example 5.1.

**Example 5.1 Software dependencies for shallow net in Keras**

Click here to view code image

```
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from matplotlib import pyplot as plt
```

We import Keras because that's the library we're using to fashion our neural network. We also import the MNIST dataset because these, of course, are the data we're working with in this example. The lines ending in `Sequential`, `Dense`, and `SGD` will make sense later; no need to worry about them at this stage. Finally, the `matplotlib` line will enable us to plot MNIST digits to our screen.

With these dependencies imported, we can conveniently load the MNIST data in a single line of code, as in Example 5.2.

**Example 5.2 Loading MNIST data**

Click here to view code image

```
(X_train, y_train), (X_valid, y_valid) = mnist.load_data()
```

Let's examine these data. As mentioned in Chapter 4, the mathematical notation *x* is used to represent the data we're feeding into a model as input, while *y* is used for the labeled output that we're training the model to predict. With this in mind, `X_train` stores the MNIST digits we'll be training our model on. [9] Executing `X_train.shape` yields the output `(60000, 28, 28)`. This shows us that, as expected, we have 60,000 images in our training dataset, each of which is a 28×28 matrix of values. Running `y_train.shape`, we unsurprisingly discover we have 60,000 labels indicating what digit is contained in each of the 60,000 training images. `y_train[0:12]` outputs an array of 12 integers representing the first dozen labels, so we can see that the first handwritten digit in the training set (`X_train[0]`) is the number *five*, the second is a *zero*, the third is a *four*, and so on.

9 . The convention is to use an uppercase letter like *X* when the variable being represented is a two-dimensional matrix or a data structure with even higher dimensionality. In contrast, a lowercase letter like *x* is used to represent a single value (a scalar) or a one-dimensional array.

Click here to view code image

```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5], dtype=uint8)
```

These happen to be the same dozen MNIST digits that were shown earlier in Figure 5.1, a figure we created by running the following chunk of code:

Click here to view code image

```
plt.figure(figsize=(5,5))
for k in range(12):
    plt.subplot(3,4, k+1)
    plt.imshow(X_train[k], cmap='Greys')
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Akin to the training data, by examining the shape of the validation data (`X_valid.shape`, `y_valid.shape`), we note that there are the expected 10,000 28×28-pixel validation images, each with a corresponding label: `(10000, 28, 28)`, `(10000,)`. Investigating the values that make up an individual image such as `X_valid[0]`, we observe that the matrix of integers representing the handwriting is primarily zeros (white-space). Tilting your head, you might even be able to make out that

the digit in this example is a *seven* with the highest integers (e.g., `254`, `255`) representing the black core of the handwritten figure, and the outline of the figure (composed of intermediate integers) fading toward white. To corroborate that this is indeed the number *seven*, we both printed out the image using `plt.imshow(X_valid[0], cmap='Greys')` (output shown in Figure 5.5) and printed out its label using `y_valid[0]` (output was 7).

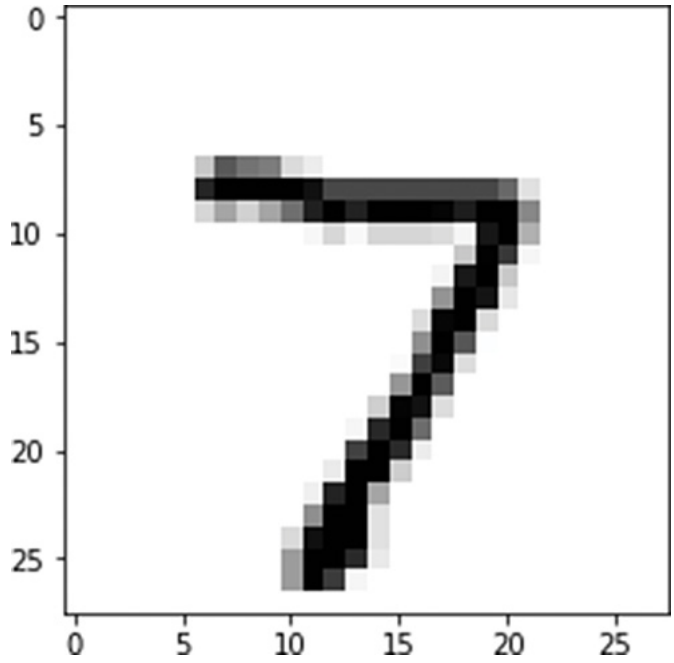


**Figure 5.5** The first MNIST digit in the validation dataset (`X_valid[0]`) is a *seven*.

## Reformatting the Data

The MNIST data now loaded, we come across the heading "Preprocess data" in the notebook. We won't, however, be preprocessing the images by applying functions to, say, extract features that provide hints to our artificial neural network. Instead, we will simply be rearranging the *shape* of the data so that they match up with the shapes of the input and output layers of the network.

Thus, we'll flatten our 28×28-pixel images into 784-element arrays. We employ the `reshape()` method, as shown in Example 5.3.

**Example 5.3 Flattening two-dimensional images to one dimension**

Click here to view code image

```
X_train = X_train.reshape(60000, 784).astype('float32')
X_valid = X_valid.reshape(10000, 784).astype('float32')
```

Simultaneously, we use `astype('float32')` to convert the pixel darknesses from integers into single-precision float values.[10] This conversion is preparation for the subsequent step, shown in Example 5.4, in which we divide all of the values by 255 so that they range from 0 to 1.[11]

10. The data are initially stored as `uint8`, which is an unsigned integer from 0 to 255. This is more memory efficient, but it doesn't require much precision because there are only 256 possible values. Without specifying, Python would default to a 64-bit float, which would be overkill. Thus, by specifying a 32-bit float we can deliberately specify a lower-precision float that is sufficient for this use case.

11. Machine learning models tend to learn more efficiently when fed standardized inputs. Binary inputs would typically be a `0` or a `1`, whereas distributions are often normalized to have a mean of 0 and a standard deviation of 1. As we've done here, pixel intensities are generally scaled to range from 0 to 1.

**Example 5.4 Converting pixel integers to floats**

```
X_train /= 255
X_valid /= 255
```

Revisiting our example handwritten *seven* from Figure 5.5 by running `X_valid[0]`, we can verify that it is now represented by a one-dimensional array made up of float values as low as 0 and as high as 1.

That's all for reformatting our model inputs *X*. As shown in Example 5.5, for the labels *y*, we need to convert them from integers into one-hot encodings (shortly we demonstrate what these are via a hands-on example).

**Example 5.5 Converting integer labels to one-hot**

Click here to view code image

```
n_classes = 10
y_train = keras.utils.to_categorical(y_train, n_classes)
y_valid = keras.utils.to_categorical(y_valid, n_classes)
```

There are 10 possible handwritten digits, so we set `n_classes` equal to `10`. In the other two lines of code we use a convenient utility function—`to_categorical`, which is provided within the Keras library—to transform both the training and the validation labels from integers into the one-hot format. Execute `y_valid` to see how the label *seven* is represented now:

Click here to view code image

```
array([0., 0., 0., 0., 0., 0., 0., 1., 0., 0.], dtype=float32)
```

Instead of using an integer to represent *seven*, we have an array of length 10 consisting entirely of `0`s, with the exception of a `1` in the eighth position. In such a one-hot encoding, the label *zero* would be represented by a lone `1` in the first position, *one* by a lone `1` in the second position, and so on. We arrange the labels with such one-hot encodings so that they line up with the 10 probabilities being output by the final layer of our artificial neural network. They represent the ideal output that we are striving to attain with our network: If the input image is a handwritten *seven*, then a perfectly trained network would output a probability of 1.00 that it is a *seven* and a probability of 0.00 for each of the other nine classes of digits.

## Designing a Neural Network Architecture

From your authors' perspective, this is the most pleasurable bit of any script featuring deep learning code: architecting the artificial neural net itself. There are infinite possibilities here, and, as you progress through the book, you will begin to develop an intuition that guides the selection of the architectures you might experiment with for tackling a given problem. Referring to Figure 5.4, for the time being, we're keeping the architecture as elementary as possible in Example 5.6.

**Example 5.6 Keras code to architect a shallow neural network**

Click here to view code image

```
model = Sequential()
model.add(Dense(64, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

In the first line of code, we instantiate the simplest type of neural network model object, the `Sequential` type[12] and—in a dash of extreme creativity—name the model `model`. In the second line, we use the `add()` method of our `model` object to specify the attributes of our network's hidden layer (`64` sigmoid-type artificial neurons in the general-purpose, fully connected arrangement defined by the `Dense()` method)[13] as well as the shape of our input layer (one-dimensional array of length `784`). In the third and final line we use the `add()` method again to specify the output layer and its parameters: `10` artificial neurons of the `softmax` variety, corresponding to the 10 probabilities (one for each of the 10 possible digits) that the network will output when fed a given handwritten image.

12. So named because each layer in the network passes information to only the next layer in the *sequence* of layers.

13. Once more, these esoteric terms will become comprehensible over the coming chapters.

## Training a Neural Network Model

Later, we return to the `model.summary()` and `model.compile()` steps of the *Shallow Net in Keras* notebook, as well as its three lines of arithmetic. For now, we skip ahead to the model-fitting step (shown in Example 5.7).

**Example 5.7 Keras code to train our shallow neural network**

Click here to view code image

```
model.fit(X_train, y_train,
          batch_size=128, epochs=200,
          verbose=1,
          validation_data=(X_valid, y_valid))
```

The critical aspects are:

1. The `fit()` method of our `model` object enables us to train our artificial neural network with the training images `X_train` as inputs and their associated labels `y_train` as the desired outputs.

2. As the network trains, the `fit()` method also provides us with the option to evaluate the performance of our network by passing our validation data `X_valid` and `y_valid` into the `validation_data` argument.

3. With machine learning, and especially with deep learning, it is commonplace to train our model on the same data multiple times. One pass through all of our training data (60,000 images in the current case) is called one *epoch* of training. By setting the `epochs` parameter to `200`, we cycle through all 60,000 training images 200 separate times.

4. By setting `verbose` to `1`, the `model.fit()` method will provide us with plenty of feedback as we train. At the moment, we'll focus on the `val_acc` statistic that is output following each epoch of training. *Validation accuracy* is the proportion of the 10,000 handwritten images in `X_valid` in which the network's highest probability in the output layer corresponds to the correct digit as per the labels in `y_valid`.

Following the first epoch of training, we observe that `val_acc` equals `0.1010`.[14, 15] That is, 10.1 percent of the images from the held-out validation dataset were correctly classified by our shallow architecture. Given that there are 10 classes of handwritten digits, we'd expect a random process to guess 10 percent of the digits correctly by chance, so this is not an impressive result. As the network continues to train, however, the results improve. After 10 epochs of training, it is correctly classifying 36.5 percent of the validation images—far better than would be expected by chance! And this is only the beginning: After 200 epochs, the network's improvement appears to be plateauing as it approaches 86 percent validation accuracy. Because we constructed an uninvolved, shallow neural-network architecture, this is not too shabby!

14. Artificial neural networks are *stochastic* (because of the way they're initialized as well as the way they learn), so your results will vary slightly from ours. Indeed, if you rerun the whole notebook (e.g., by clicking on the *Kernel* option in the Jupyter menu bar and selecting *Restart & Run All*), you should obtain new, slightly different results each time you do this.

15. By the end of Chapter 8, you'll have enough theory under your belt to study the output `model.fit()` in all its glory. For our immediate "cart before the horse" purposes, coverage of the *validation accuracy* metric alone suffices.

## SUMMARY

Putting the cart before the horse, in this chapter we coded up a shallow, elementary artificial neural network. With decent accuracy, it is able to classify the MNIST images. Over the remainder of Part II, as we dive into theory, unearth artificial-neural-network best practices, and layer up to authentic deep learning architectures, we should surely be able to classify inputs much more accurately, no? Let's see.

Settings / Support / Sign Out

PREV
II: Essential Theory Illustrated

NEXT
6. Artificial Neurons Detecting Hot Dogs