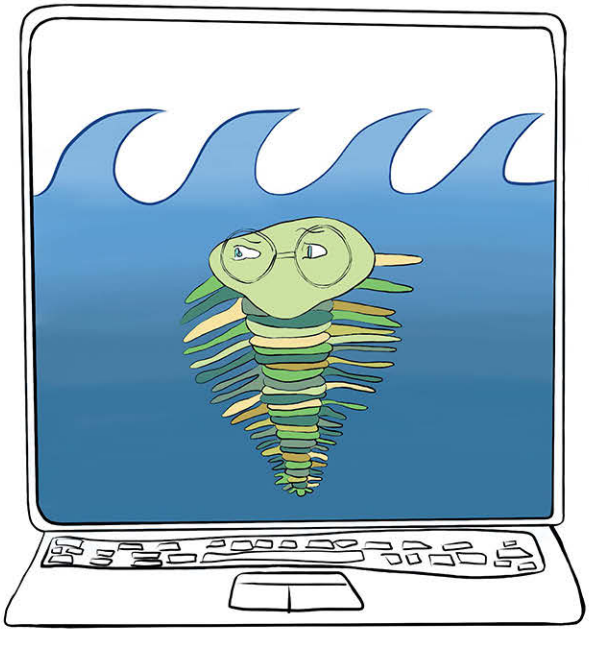
ADDISON WESLEY DATA & ANALYTICS SERIES

DEEP LEARNING ILLUSTRATED

A Visual, Interactive Guide to Artificial Intelligence



JON KROHN

with **GRANT BEYLEVELD** *and* **AGLAÉ BASSENS**

Praise for *Deep Learning Illustrated*

"Over the next few decades, artificial intelligence is poised to dramatically change almost every aspect of our lives, in large part due to today's breakthroughs in deep learning. The authors' clear visual style provides a comprehensive look at what's currently possible with artificial neural networks as well as a glimpse of the magic that's to come."
—*Tim Urban, writer and illustrator of Wait But Why*

"This book is an approachable, practical, and broad introduction to deep learning, and the most beautifully illustrated machine learning book on the market."
—*Dr. Michael Osborne, Dyson Associate Professor in Machine Learning, University of Oxford*

"This book should be the first stop for deep learning beginners, as it contains lots of concrete, easy-to-follow examples with corresponding tutorial videos and code notebooks. Strongly recommended."
—*Dr. Chong Li, cofounder, Nakamoto & Turing Labs; adjunct professor, Columbia University*

"It's hard to imagine developing new products today without thinking about enriching them with capabilities using machine learning. Deep learning in particular has many practical applications, and this book's intelligible clear and visual approach is helpful to anyone who would like to understand what deep learning is and how it could impact your business and life for years to come."
—*Helen Altshuler, engineering leader, Google*

Example 5.3 Flattening two-dimensional images to one dimension

```
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]

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

```
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:

```
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

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.

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

```
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.

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

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

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.

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!

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.

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.

6

Artificial Neurons Detecting Hot Dogs

Having received tantalizing exposure to applications of deep learning in the first part of this book and having coded up a functioning neural network in Chapter 5, the moment has come to delve into the nitty-gritty theory underlying these capabilities. We begin by dissecting artificial neurons, the units that—when wired together—constitute an artificial neural network.

Biological Neuroanatomy 101

As presented in the opening paragraphs of this book, ersatz neurons are inspired by biological ones. Given that, let's take a gander at Figure 6.1 for a précis of the first lecture in any neuroanatomy course: A given biological neuron receives input into its *cell body* from many (generally thousands) of *dendrites*, with each dendrite receiving signals of information from another neuron in the nervous system—a biological neural network. When the signal conveyed along a dendrite reaches the cell body, it causes a small change in the voltage of the cell body.[1] Some dendrites cause a small positive change in voltage, and the others cause a small negative change. If the cumulative effect of these changes causes the voltage to increase from its resting state of −70 millivolts to the critical threshold of −55 millivolts, the neuron will fire something called an *action potential* away from its cell body, down its axon, thereby transmitting a signal to other neurons in the network.

To summarize, biological neurons exhibit the following three behaviors in sequence:

1. Receive information from many other neurons
2. Aggregate this information via changes in cell voltage at the cell body
3. Transmit a signal if the cell voltage crosses a threshold level, a signal that can be received by many other neurons in the network

1. More precisely, it causes a change in the voltage *difference* between the cell's interior and its surroundings.

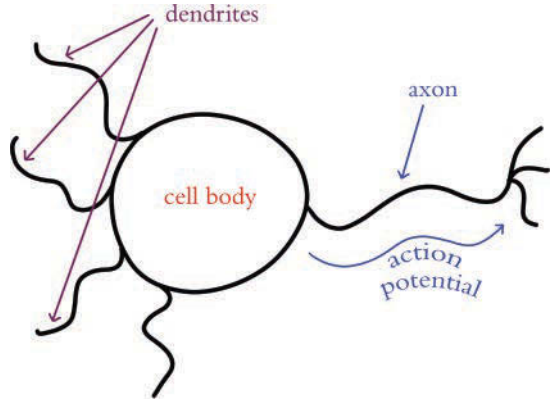


Figure 6.1 The anatomy of a biological neuron

We've aligned the purple, red, and blue colors of the text here with the colors (indicating dendrites, cell body, and the axon, respectively) in Figure 6.1. We'll do this time and again throughout the book, including to discuss key equations and the variables they contain.

The Perceptron

In the late 1950s, the American neurobiologist Frank Rosenblatt (Figure 6.2) published an article on his *perceptron*, an algorithm influenced by his understanding of biological neurons, making it the earliest formulation of an artificial neuron.[2] Analogous to its living inspiration, the perceptron (Figure 6.3) can:

1. Receive input from multiple other neurons
2. Aggregate those inputs via a simple arithmetic operation called the *weighted sum*
3. Generate an output if this weighted sum crosses a threshold level, which can then be sent on to many other neurons within a network

The Hot Dog / Not Hot Dog Detector

Let's work through a lighthearted example to understand how the perceptron algorithm works. We're going to look at a perceptron that is specialized in distinguishing whether a given object is a hot dog or, well . . . not a hot dog.

A critical attribute of perceptrons is that they can only be fed binary information as inputs, and their output is also restricted to being binary. Thus, our hot dog-detecting

2. Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and the organization in the brain. *Psychological Review, 65,* 386–408.

Figure 6.2 The American neurobiology and behavior researcher Frank Rosenblatt. He conducted much of his work out of the Cornell Aeronautical Laboratory, including physically constructing his Mark I Perceptron there. This machine, an early relic of artificial intelligence, can today be viewed at the Smithsonian Institution in Washington, D.C.

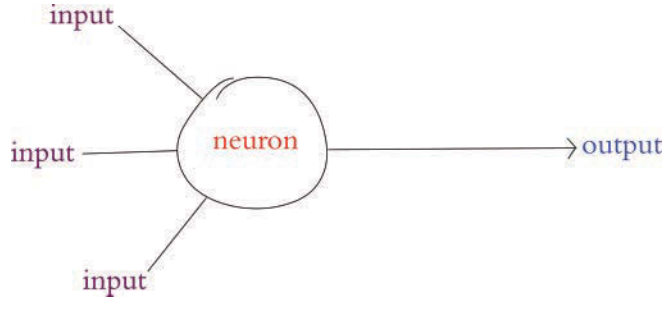


Figure 6.3 Schematic diagram of a perceptron, an early artificial neuron. Note the structural similarity to the biological neuron in Figure 6.1.

perceptron must be fed its particular three inputs (indicating whether the object involves ketchup, mustard, or a bun, respectively) as either a 0 or a 1. In Figure 6.4:

- The first input (a purple 1) indicates the object being presented to the perceptron involves ketchup.
- The second input (also a purple 1) indicates the object has mustard.
- The third input (a purple 0) indicates the object does *not* include a bun.

To make a prediction as to whether the object is a hot dog or not, the perceptron independently *weights* each of these three inputs.[3] The weights that we arbitrarily selected

3. If you are well accustomed to regression modeling, this should be a familiar paradigm.