**PART II**

**Neural Networks and Deep Learning**



**CHAPTER 10**

**Introduction to Arti€cial Neural Networks with Keras**



With Early Release ebooks, you get books in their earliest form— the author’s raw and unedited content as he or she writes—so you can take advantage of these technologies long before the official release of these titles. The following will be Chapter 10 in the final release of the book.



Birds inspired us to fly, burdock plants inspired velcro, and countless more inven‐ tions were inspired by nature. It seems only logical, then, to look at the brain’s archi‐ tecture for inspiration on how to build an intelligent machine. This is the key idea that sparked *artificial neural networks* (ANNs). However, although planes were inspired by birds, they don’t have to flap their wings. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying “units” rather than “neurons”), lest we restrict our creativity to biologically plausible systems.[1](#page3)

ANNs are at the very core of Deep Learning. They are versatile, powerful, and scala‐ ble, making them ideal to tackle large and highly complex Machine Learning tasks, such as classifying billions of images (e.g., Google Images), powering speech recogni‐ tion services (e.g., Apple’s Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of *Go* by playing millions of games against itself (DeepMind’s Alpha‐ Zero).



1. You can get the best of both worlds by being open to biological inspirations without being afraid to create biologically unrealistic models, as long as they work well.



**277**

In the first part of this chapter, we will introduce artificial neural networks, starting with a quick tour of the very first ANN architectures, leading up to *Multi-Layer Per‐* *ceptrons* (MLPs) which are heavily used today (other architectures will be explored inthe next chapters). In the second part, we will look at how to implement neural net‐ works using the popular Keras API. This is a beautifully designed and simple high-level API for building, training, evaluating and running neural networks. But don’t be fooled by its simplicity: it is expressive and flexible enough to let you build a wide variety of neural network architectures. In fact, it will probably be sufficient for most of your use cases. Moreover, should you ever need extra flexibility, you can always write custom Keras components using its lower-level API, as we will see in Chap‐ ter 12.

But first, let’s go back in time to see how artificial neural networks came to be!

**From Biological to Arti€cial Neurons**

Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their [landmark paper](https://homl.info/43),[2](#page4) “A Logical Calculus of Ideas Immanent in Nervous Activity,” McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using *propositional logic*. This was the first artificial neural network architecture. Since then many other architectures have been invented, as we will see.

The early successes of ANNs until the 1960s led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere and ANNs entered a long winter. In the early 1980s there was a revival of interest in *connectionism* (the study of neural networks), as new architectures were invented andbetter training techniques were developed. But progress was slow, and by the 1990s other powerful Machine Learning techniques were invented, such as Support Vector Machines (see Chapter 5). These techniques seemed to offer better results and stron‐ ger theoretical foundations than ANNs, so once again the study of neural networks entered a long winter.

Finally, we are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, there are a few good reasons to believe that this wave is different and that it will have a much more profound impact on our lives:



2 “A Logical Calculus of Ideas Immanent in Nervous Activity,” W. McCulloch and W. Pitts (1943).



**278** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

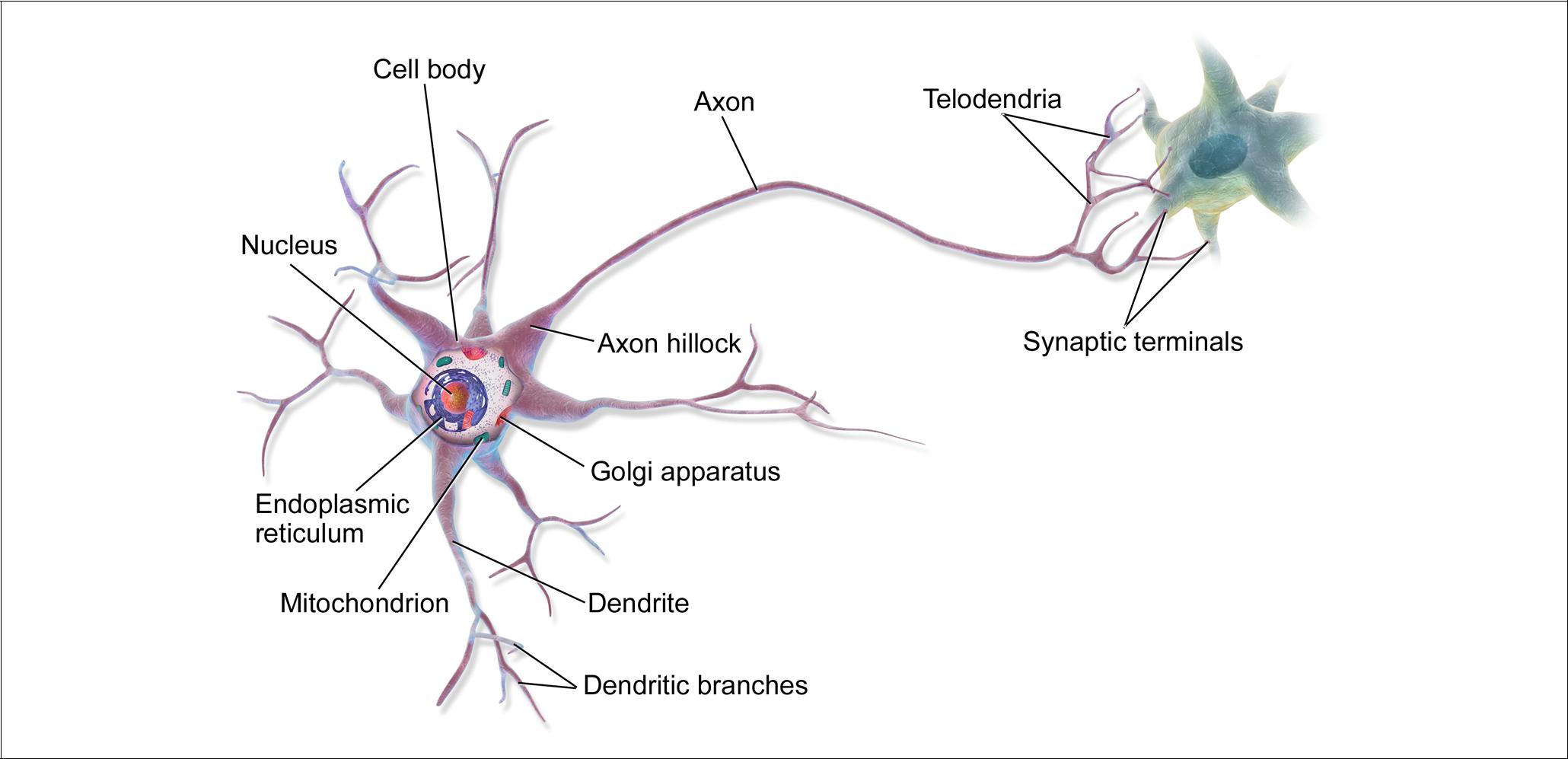
* There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
* The tremendous increase in computing power since the 1990s now makes it pos‐ sible to train large neural networks in a reasonable amount of time. This is in part due to Moore’s Law, but also thanks to the gaming industry, which has pro‐ duced powerful GPU cards by the millions.
* The training algorithms have been improved. To be fair they are only slightly dif‐ ferent from the ones used in the 1990s, but these relatively small tweaks have a huge positive impact.
* Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is rather rare in practice (or when it is the case, they are usually fairly close to the global optimum).
* ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more pro‐ gress, and even more amazing products.

**Biological Neurons**

Before we discuss artificial neurons, let’s take a quick look at a biological neuron (rep‐ resented in [Figure 10-1](#page6)). It is an unusual-looking cell mostly found in animal cerebral cortexes (e.g., your brain), composed of a *cell body* containing the nucleus and most of the cell’s complex components, and many branching extensions called *dendrites*, plus one very long extension called the *axon*. The axon’s length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called *synaptic terminals* (or simply *synap‐* *ses*), which are connected to the dendrites (or directly to the cell body) of other neu‐rons. Biological neurons receive short electrical impulses called *signals* from other neurons via these synapses. When a neuron receives a sufficient number of signals from other neurons within a few milliseconds, it fires its own signals.

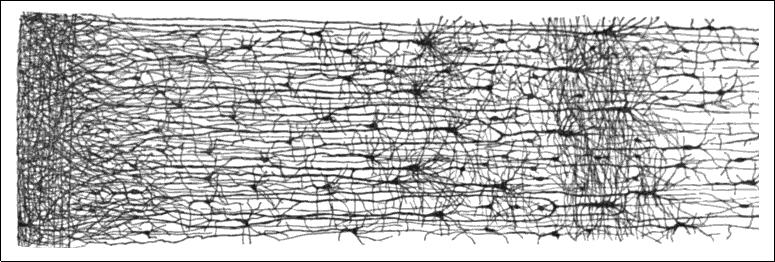


**From Biological to Arti€cial Neurons** **|** **279**



*Figure 10-1. Biological neuron*[*3*](#page6)

Thus, individual biological neurons seem to behave in a rather simple way, but they are organized in a vast network of billions of neurons, each neuron typically connec‐ ted to thousands of other neurons. Highly complex computations can be performed by a vast network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural net‐ works (BNN)[4](#page6) is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, as shown in [Figure 10-2](#page6).



*Figure 10-2. Multiple layers in a biological neural network (human cortex)*[*5*](#page6)

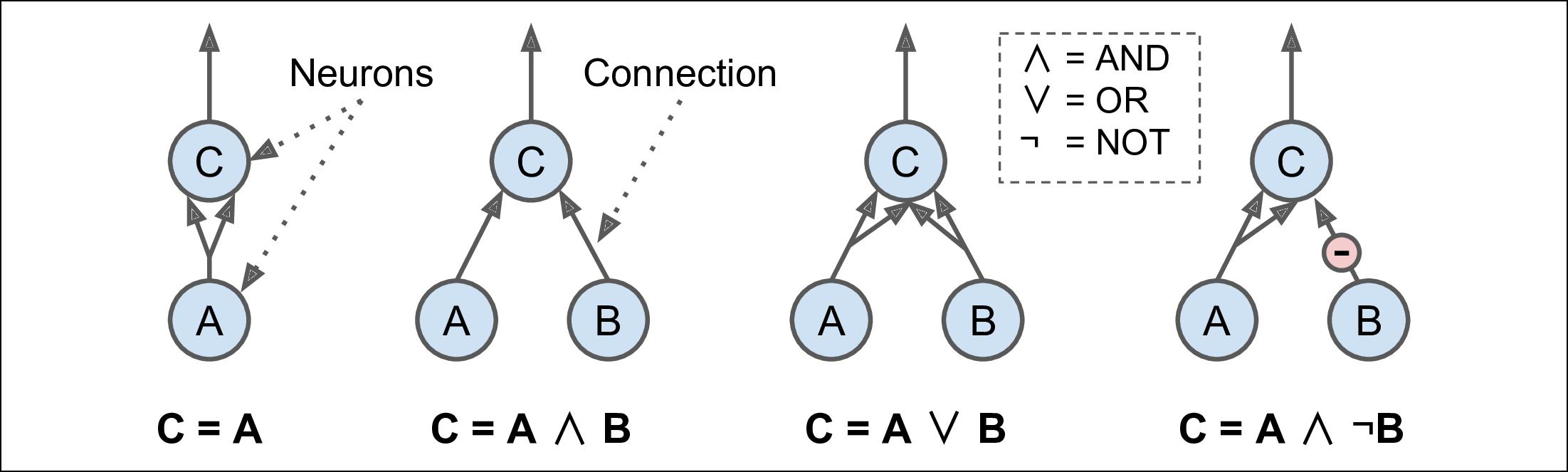


1. Image by Bruce Blaus ([Creative Commons 3.0](https://creativecommons.org/licenses/by/3.0/)). Reproduced from [*https://en.wikipedia.org/wiki/Neuron*](https://en.wikipedia.org/wiki/Neuron).
2. In the context of Machine Learning, the phrase “neural networks” generally refers to ANNs, not BNNs.
3. Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from [*https://en.wikipe*](https://en.wikipedia.org/wiki/Cerebral_cortex) [*dia.org/wiki/Cerebral\_cortex*](https://en.wikipedia.org/wiki/Cerebral_cortex).
4. 

**280** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

**Logical Computations with Neurons**

Warren McCulloch and Walter Pitts proposed a very simple model of the biological neuron, which later became known as an *artificial neuron*: it has one or more binary (on/off) inputs and one binary output. The artificial neuron simply activates its out‐ put when more than a certain number of its inputs are active. McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that computes any logical proposition you want. For example, let’s build a few ANNs that perform various logical computations (see [Figure 10-3](#page7)), assuming that a neuron is activated when at least two of its inputs are active.



*Figure 10-3. ANNs performing simple logical computations*

* The first network on the left is simply the identity function: if neuron A is activa‐ ted, then neuron C gets activated as well (since it receives two input signals from neuron A), but if neuron A is off, then neuron C is off as well.
* The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to acti‐ vate neuron C).
* The third network performs a logical OR: neuron C gets activated if either neu‐ ron A or neuron B is activated (or both).
* Finally, if we suppose that an input connection can inhibit the neuron’s activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron A is active and if neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can easily imagine how these networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter).

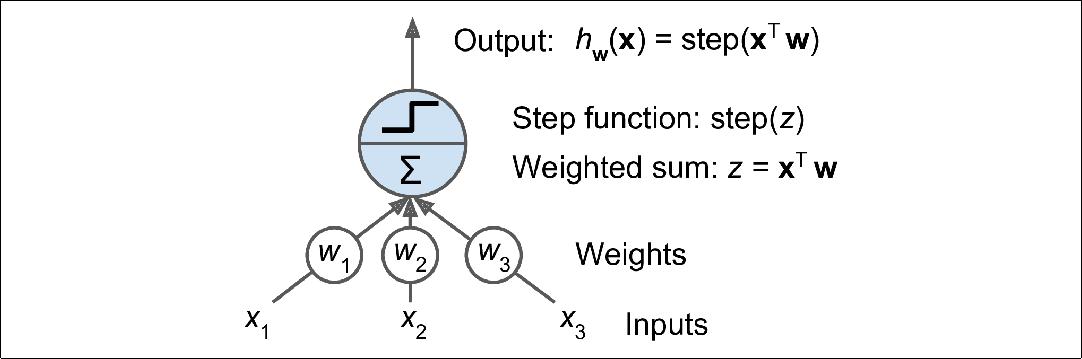
**The Perceptron**

The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron (see [Figure 10-4](#page8)) called



**From Biological to Arti€cial Neurons** **|** **281**

a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU): the inputs and output are now numbers (instead of binary on/off values) and each input con‐ nection is associated with a weight. The TLU computes a weighted sum of its inputs (*z* = *w*1 *x*1 + *w*2 *x*2 + ⋯ + *wn* *xn* = **x***T* **w**), then applies a *step function* to that sum and outputs the result: *h***w**(**x**) = step(*z*), where *z* = **x***T* **w**.



*Figure 10-4. Threshold logic unit*

The most common step function used in Perceptrons is the *Heaviside step function* (see [Equation 10-1](#page8)). Sometimes the sign function is used instead.

*Equation 10-1. Common step functions used in Perceptrons*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | 0 | if | *z* < 0 | −1 if *z* < 0 |  |
| heaviside *z* | = | sgn *z* = 0 if *z* = 0 |  |
| 1 | if | *z* ≥ 0 |  |
|  |  |  |  |  | +1 if *z* > 0 |  |



A single TLU can be used for simple linear binary classification. It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a Logistic Regression classifier or a linear SVM). For example, you could use a single TLU to classify iris flowers based on the petal length and width (also adding an extra bias feature *x*0 = 1, just like we did in previous chapters). Training a TLU in this case means finding the right values for *w*0, *w*1, and *w*2(the training algorithm is discussed shortly).

A Perceptron is simply composed of a single layer of TLUs,[6](#page8) with each TLU connected to all the inputs. When all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), it is called a *fully connected layer* or a *dense* *layer*. To represent the fact that each input is sent to every TLU, it is common to drawspecial passthrough neurons called *input neurons*: they just output whatever input they are fed. All the input neurons form the *input layer*. Moreover, an extra bias fea‐

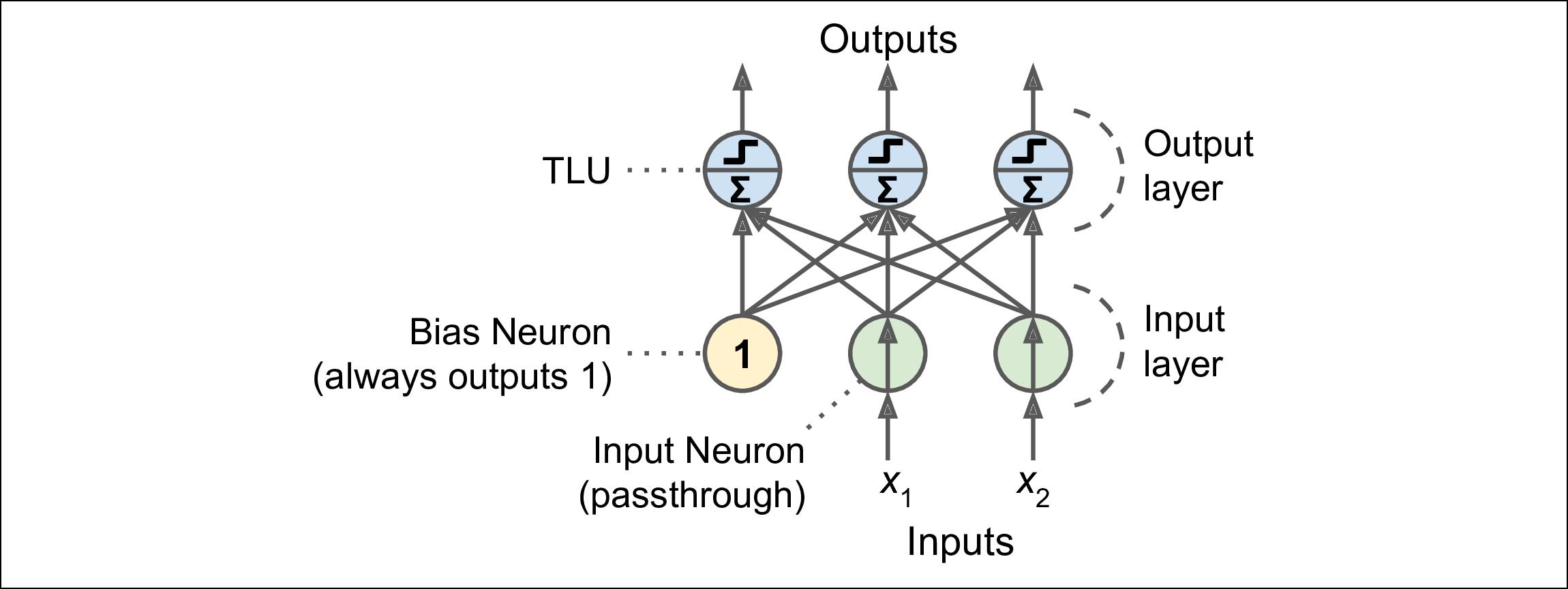


6 The name *Perceptron* is sometimes used to mean a tiny network with a single TLU.



**282** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

ture is generally added (*x*0 = 1): it is typically represented using a special type of neu‐ ron called a *bias neuron*, which just outputs 1 all the time. A Perceptron with two inputs and three outputs is represented in [Figure 10-5](#page9). This Perceptron can classify instances simultaneously into three different binary classes, which makes it a multi‐ output classifier.



*Figure 10-5. Perceptron diagram*

Thanks to the magic of linear algebra, it is possible to efficiently compute the outputs of a layer of artificial neurons for several instances at once, by using [Equation 10-2](#page9):

*Equation 10-2. Computing the outputs of a fully connected layer h***W**,**b****X** = *ϕ***XW** + **b**

* As always, **X** represents the matrix of input features. It has one row per instance, one column per feature.
* The weight matrix **W** contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artifi‐ cial neuron in the layer.
* The bias vector **b** contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
* The function *ϕ* is called the *activation function*: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

So how is a Perceptron trained? The Perceptron training algorithm proposed by Frank Rosenblatt was largely inspired by *Hebb’s rule*. In his book *The Organization of* *Behavior*, published in 1949, Donald Hebb suggested that when a biological neuronoften triggers another neuron, the connection between these two neurons grows stronger. This idea was later summarized by Siegrid Löwel in this catchy phrase: “Cells that fire together, wire together.” This rule later became known as Hebb’s rule



**From Biological to Arti€cial Neurons** **|** **283**

(or *Hebbian learning*); that is, the connection weight between two neurons is increased whenever they have the same output. Perceptrons are trained using a var‐ iant of this rule that takes into account the error made by the network; it reinforces connections that help reduce the error. More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in [Equation 10-3](#page10).

*Equation 10-3. Perceptron learning rule (weight update)*

*wi*, *j*next step = *wi*, *j* + *η**yj* − *y j**xi*

* *wi*, *j* is the connection weight between the *i*thinput neuron and the *j*thoutput neu‐ron.
* *xi* is the *i*thinput value of the current training instance.
* *y j* is the output of the *j*thoutput neuron for the current training instance.
* *yj* is the target output of the *j*thoutput neuron for the current training instance.
* *η* is the learning rate.

The decision boundary of each output neuron is linear, so Perceptrons are incapable of learning complex patterns (just like Logistic Regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.[7](#page10) This is called the *Perceptron convergence theorem*.

Scikit-Learn provides a Perceptron class that implements a single TLU network. It can be used pretty much as you would expect—for example, on the iris dataset (intro‐ duced in Chapter 4):

**import numpy as np**

**from sklearn.datasets import** load\_iris **from sklearn.linear\_model import** Perceptron

iris = load\_iris()

X = iris.data[:, (2, 3)] *# petal length, petal width* y = (iris.target == 0).astype(np.int) *# Iris Setosa?*

per\_clf = Perceptron() per\_clf.fit(X, y)



1. Note that this solution is generally not unique: in general when the data are linearly separable, there is an infinity of hyperplanes that can separate them.



**284** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

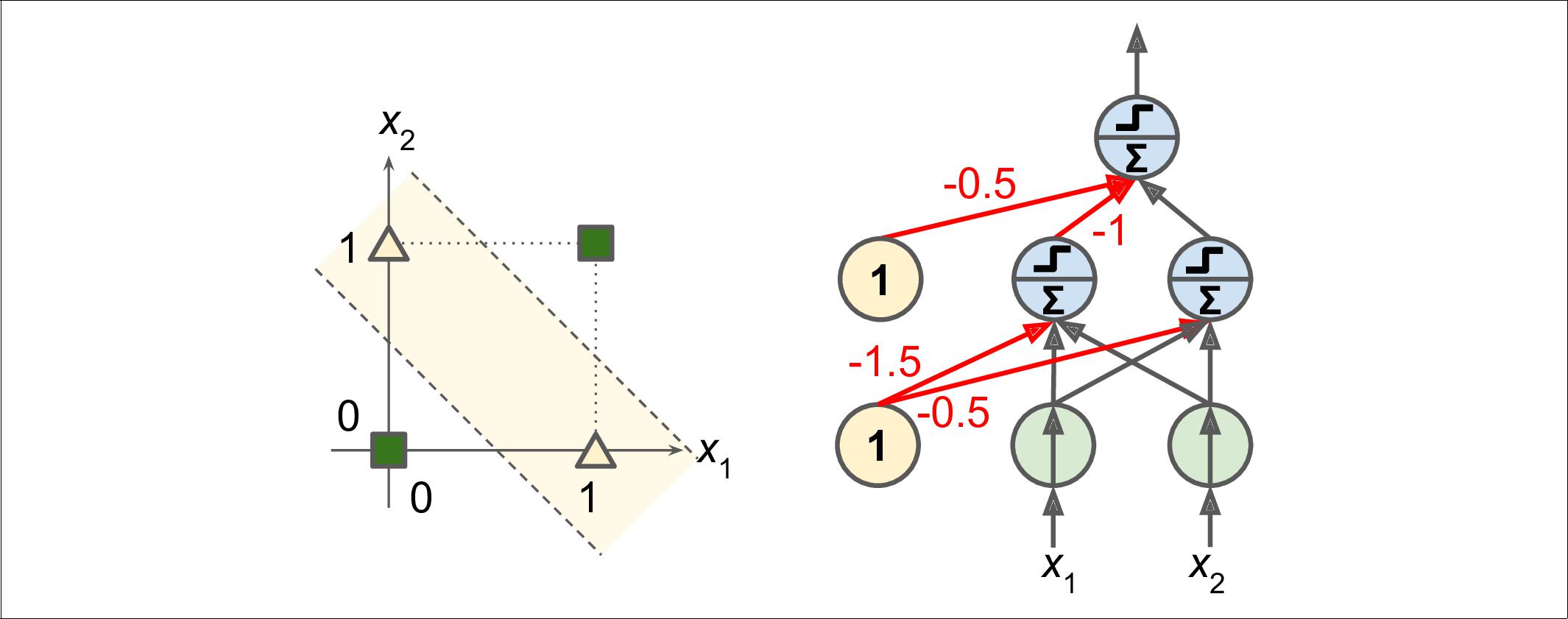
y\_pred = per\_clf.predict([[2, 0.5]])

You may have noticed the fact that the Perceptron learning algorithm strongly resem‐ bles Stochastic Gradient Descent. In fact, Scikit-Learn’s Perceptron class is equivalent to using an SGDClassifier with the following hyperparameters: loss="perceptron", learning\_rate="constant", eta0=1 (the learning rate), and penalty=None (no regu‐ larization).

Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they just make predictions based on a hard threshold. This is one of the good reasons to prefer Logistic Regression over Perceptrons.

In their 1969 monograph titled *Perceptrons*, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of Perceptrons, in particular the fact that they are incapable of solving some trivial problems (e.g., the *Exclusive OR* (XOR) classification problem; see the left side of [Figure 10-6](#page11)). Of course this is true of any other linear classification model as well (such as Logistic Regression classifiers), but researchers had expected much more from Perceptrons, and their disappointment was great, and many researchers dropped neural networks altogether in favor of higher-level problems such as logic, problem solving, and search.

However, it turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a *Multi-Layer Perceptron* (MLP). In particular, an MLP can solve the XOR problem, as you can verify by com‐ puting the output of the MLP represented on the right of [Figure 10-6](#page11): with inputs (0, 0) or (1, 1) the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. All connections have a weight equal to 1, except the four connections where the weight is shown. Try verifying that this network indeed solves the XOR problem!



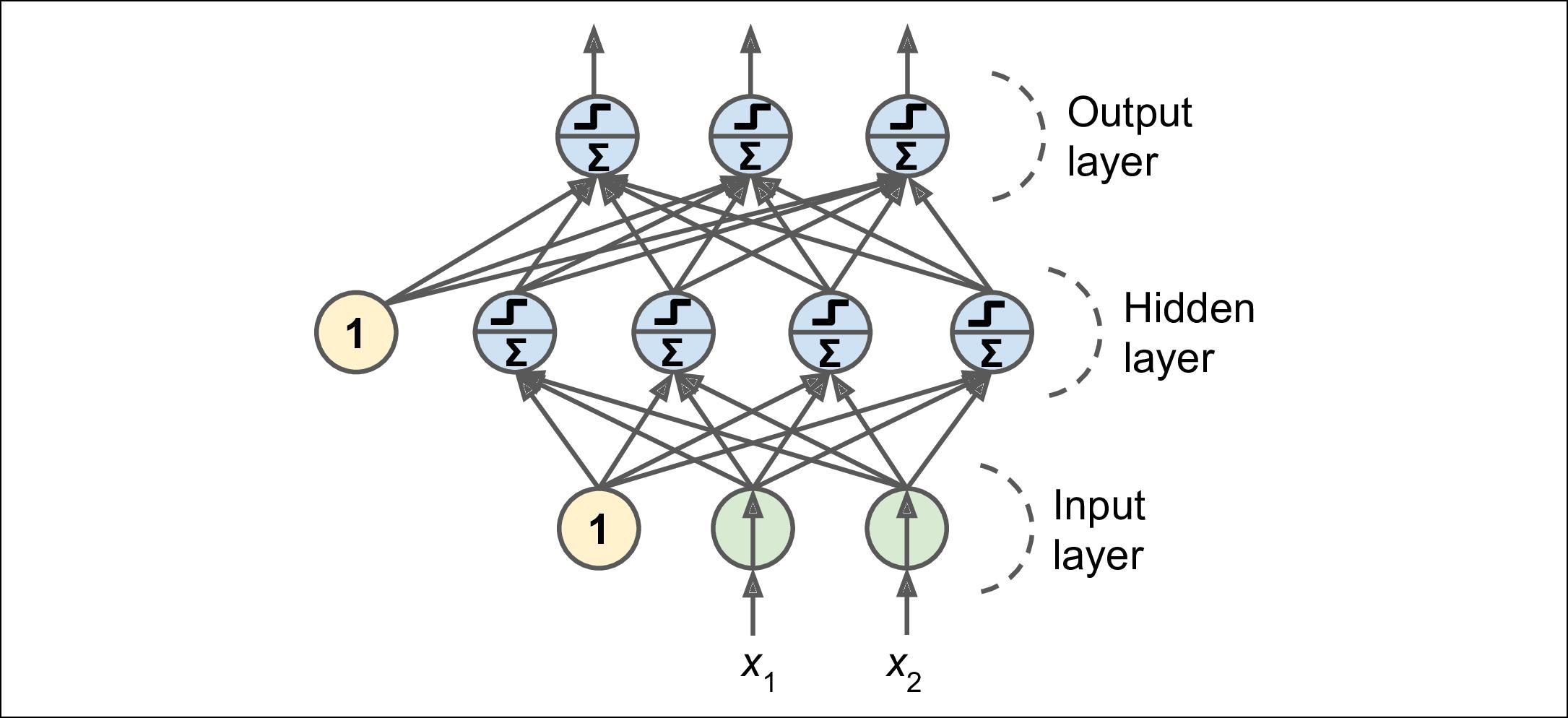
*Figure 10-6. XOR classification problem and an MLP that solves it*



**From Biological to Arti€cial Neurons** **|** **285**

**Multi-Layer Perceptron and Backpropagation**

An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer* (see [Figure 10-7](#page12)). The layers close to the input layer are usually called the lower layers, and the ones close to the outputs are usually called the upper layers. Every layer except the output layer includes a bias neuron and is fully connected to the next layer.



*Figure 10-7. Multi-Layer Perceptron*

The signal flows only in one direction (from the inputs to the out‐ puts), so this architecture is an example of a *feedforward neural net‐* *work* (FNN).



When an ANN contains a deep stack of hidden layers[8](#page12), it is called a *deep neural net‐* *work* (DNN). The field of Deep Learning studies DNNs, and more generally modelscontaining deep stacks of computations. However, many people talk about Deep Learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton and Ronald Williams published a [groundbreaking paper](https://homl.info/44)[9](#page12) introducing the *backpropagation* training algorithm, which is still used today. In short, it is simply Gradient Descent (introduced in Chapter 4)



1. In the 1990s, an ANN with more than two hidden layers was considered deep. Nowadays, it is common to see ANNs with dozens of layers, or even hundreds, so the definition of “deep” is quite fuzzy.
2. “Learning Internal Representations by Error Propagation,” D. Rumelhart, G. Hinton, R. Williams (1986).



**286** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

using an efficient technique for computing the gradients automatically[10](#page13): in just two passes through the network (one forward, one backward), the backpropagation algo‐ rithm is able to compute the gradient of the network’s error with regards to every sin‐ gle model parameter. In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gra‐ dients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.

Automatically computing gradients is called *automatic di‰erentia‚* *tion*, or *autodi‰*. There are various autodiff techniques, with differ‐ent pros and cons. The one used by backpropagation is called *reverse-mode autodi‰*. It is fast and precise, and is well suited whenthe function to differentiate has many variables (e.g., connection weights) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out ???.



Let’s run through this algorithm in a bit more detail:

* It handles one mini-batch at a time (for example containing 32 instances each), and it goes through the full training set multiple times. Each pass is called an *epoch*, as we saw inChapter 4.
* Each mini-batch is passed to the network’s input layer, which just sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
* Next, the algorithm measures the network’s output error (i.e., it uses a loss func‐ tion that compares the desired output and the actual output of the network, and returns some measure of the error).
* Then it computes how much each output connection contributed to the error. This is done analytically by simply applying the *chain rule* (perhaps the most fun‐ damental rule in calculus), which makes this step fast and precise.
* The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule—and so on until the algorithm reaches the input layer. As we explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights in the



1. This technique was actually independently invented several times by various researchers in different fields, starting with P. Werbos in 1974.



**From Biological to Arti€cial Neurons** **|** **287**

network by propagating the error gradient backward through the network (hence the name of the algorithm).

* Finally, the algorithm performs a Gradient Descent step to tweak all the connec‐ tion weights in the network, using the error gradients it just computed.

This algorithm is so important, it’s worth summarizing it again: for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse to measure the error con‐ tribution from each connection (reverse pass), and finally slightly tweaks the connec‐ tion weights to reduce the error (Gradient Descent step).

It is important to initialize all the hidden layers’ connection weights randomly, or else training will fail. For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical. In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won’t be too smart. If instead you randomly initialize the weights, you *break the symme‐* *try* and allow backpropagation to train a diverse team of neurons.



In order for this algorithm to work properly, the authors made a key change to the MLP’s architecture: they replaced the step function with the logistic function, *σ*(*z*) = 1 / (1 + exp(–*z*)). This was essential because the step function contains only flat seg‐ ments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the logistic function has a well-defined nonzero derivative every‐ where, allowing Gradient Descent to make some progress at every step. In fact, the backpropagation algorithm works well with many other *activation functions*, not just the logistic function. Two other popular activation functions are:

*The hyperbolic tangent function tanh(z) = 2σ(2z) – 1*

Just like the logistic function it is S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1 (instead of 0 to 1 in the case of the logistic func‐ tion), which tends to make each layer’s output more or less centered around 0 at the beginning of training. This often helps speed up convergence.

*The Rectified Linear Unit function: ReLU(z) = max(0, z)*

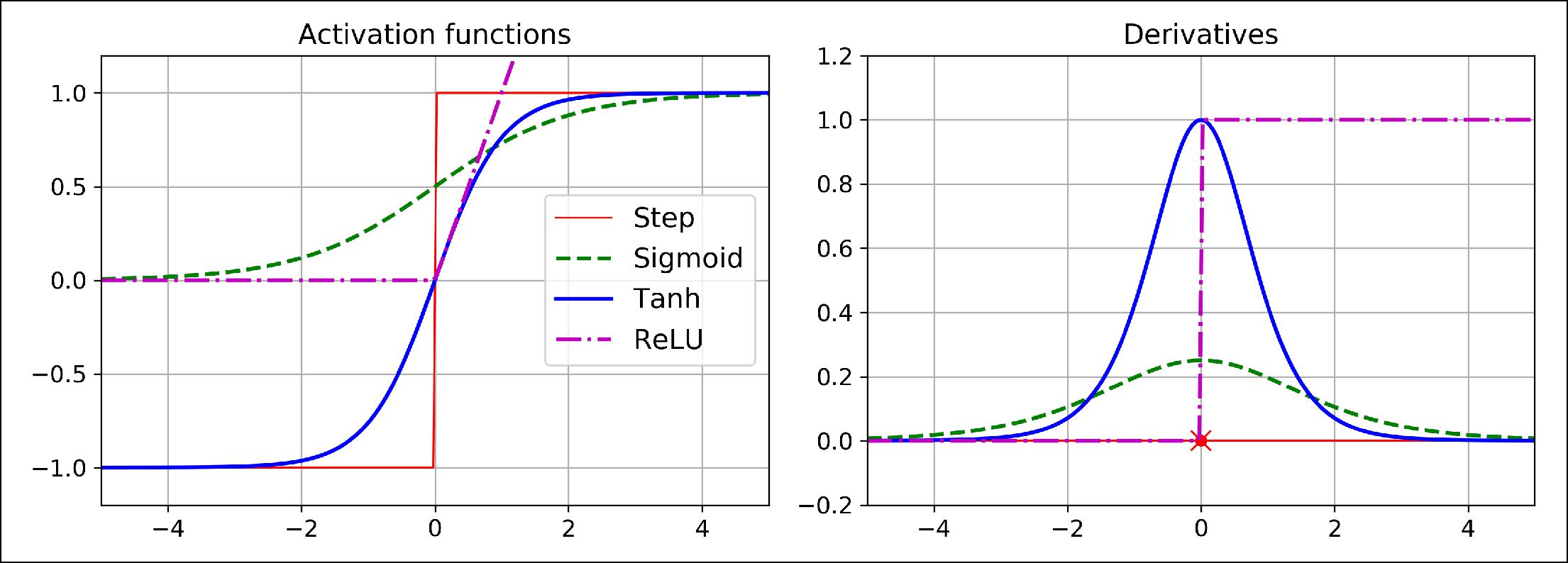
It is continuous but unfortunately not differentiable at *z* = 0 (the slope changes abruptly, which can make Gradient Descent bounce around), and its derivative is 0 for *z* < 0. However, in practice it works very well and has the advantage of being



**288** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

fast to compute[11](#page15). Most importantly, the fact that it does not have a maximum output value also helps reduce some issues during Gradient Descent (we will come back to this in Chapter 11).

These popular activation functions and their derivatives are represented in [Figure 10-8](#page15). But wait! Why do we need activation functions in the first place? Well, if you chain several linear transformations, all you get is a linear transformation. For example, say f(*x*) = 2 *x* + 3 and g(*x*) = 5 *x* - 1, then chaining these two linear functions gives you another linear function: f(g(*x*)) = 2(5 *x* - 1) + 3 = 10 *x* + 1. So if you don’t have some non-linearity between layers, then even a deep stack of layers is equivalent to a single layer: you cannot solve very complex problems with that.



*Figure 10-8. Activation functions and their derivatives*

Okay! So now you know where neural nets came from, what their architecture is and how to compute their outputs, and you also learned about the backpropagation algo‐ rithm. But what exactly can you do with them?

**Regression MLPs**

First, MLPs can be used for regression tasks. If you want to predict a single value (e.g., the price of a house given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object on an image, you need to predict 2D coordi‐ nates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So you end up with 4 output neurons.



1. Biological neurons seem to implement a roughly sigmoid (S-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that ReLU generally works better in ANNs. This is one of the cases where the biological analogy was misleading.



**From Biological to Arti€cial Neurons** **|** **289**

In general, when building an MLP for regression, you do not want to use any activa‐ tion function for the output neurons, so they are free to output any range of values. However, if you want to guarantee that the output will always be positive, then you can use the ReLU activation function, or the *so•plus* activation function in the output layer. Finally, if you want to guarantee that the predictions will fall within a given range of values, then you can use the logistic function or the hyperbolic tangent, and scale the labels to the appropriate range: 0 to 1 for the logistic function, or –1 to 1 for the hyperbolic tangent.

The loss function to use during training is typically the mean squared error, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you can use the Huber loss, which is a combination of both.



The Huber loss is quadratic when the error is smaller than a thres‐ hold δ (typically 1), but linear when the error is larger than δ. This makes it less sensitive to outliers than the mean squared error, and it is often more precise and converges faster than the mean abso‐ lute error.

[Table 10-1](#page16) summarizes the typical architecture of a regression MLP.

*Table 10-1. Typical Regression MLP Architecture*

|  |  |
| --- | --- |
| **Hyperparameter** | **Typical Value** |
| # input neurons | One per input feature (e.g., 28 x 28 = 784 for MNIST) |
| # hidden layers | Depends on the problem. Typically 1 to 5. |
| # neurons per hidden layer | Depends on the problem. Typically 10 to 100. |
| # output neurons | 1 per prediction dimension |
| Hidden activation | ReLU (or SELU, see Chapter 11) |
| Output activation | None or ReLU/Softplus (if positive outputs) or Logistic/Tanh (if bounded outputs) |
| Loss function | MSE or MAE/Huber (if outliers) |
|  |  |

**Classi€cation MLPs**

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the logistic activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probabil‐ ity of the positive class. Obviously, the estimated probability of the negative class is equal to one minus that number.

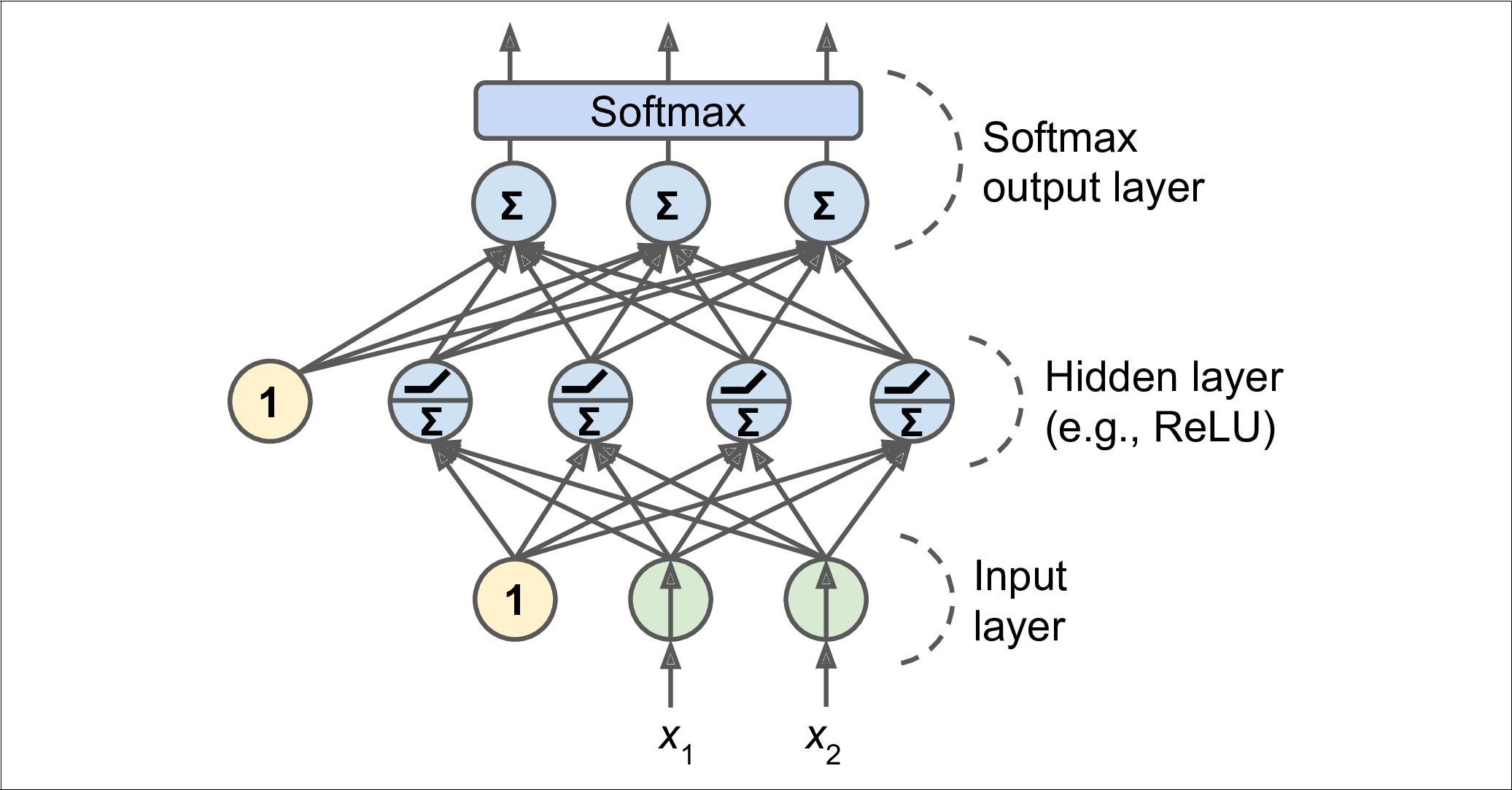
MLPs can also easily handle multilabel binary classification tasks (see Chapter 3). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent



**290** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

or non-urgent email. In this case, you would need two output neurons, both using the logistic activation function: the first would output the probability that the email is spam and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to one. This lets the model output any combi‐ nation of labels: you can have non-urgent ham, urgent ham, non-urgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of 3 or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the *so•max* activation function for the whole output layer (see [Figure 10-9](#page17)). The softmax function (introduced in Chapter 4) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to one (which is required if the classes are exclusive). This is called multiclass clas‐ sification.



*Figure 10-9. A modern MLP (including ReLU and so•max) for classification*

Regarding the loss function, since we are predicting probability distributions, the cross-entropy (also called the log loss, see Chapter 4) is generally a good choice.

[Table 10-2](#page17) summarizes the typical architecture of a classification MLP.

*Table 10-2. Typical Classification MLP Architecture*

|  |  |  |  |
| --- | --- | --- | --- |
| **Hyperparameter** | **Binary classi€cation** | **Multilabel binary classi€cation** | **Multiclass classi€cation** |
| Input and hidden layers | Same as regression | Same as regression | Same as regression |
| # output neurons | 1 | 1 per label | 1 per class |
| Output layer activation | Logistic | Logistic | Softmax |



**From Biological to Arti€cial Neurons** **|** **291**

|  |  |  |  |
| --- | --- | --- | --- |
| **Hyperparameter** | **Binary classi€cation** | **Multilabel binary classi€cation** | **Multiclass classi€cation** |
| Loss function | Cross-Entropy | Cross-Entropy | Cross-Entropy |
|  |  |  |  |



Before we go on, I recommend you go through exercise 1, at the end of this chapter. You will play with various neural network architectures and visualize their outputs using the *TensorFlow Play‐* *ground*. This will be very useful to better understand MLPs, forexample the effects of all the hyperparameters (number of layers and neurons, activation functions, and more).

Now you have all the concepts you need to start implementing MLPs with Keras!

**Implementing MLPs with Keras**

Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate and execute all sorts of neural networks. Its documentation (or specification) is avail‐ able at [*https://keras.io*](https://keras.io). The reference implementation is simply called Keras as well, so to avoid any confusion we will call it keras-team (since it is available at [*https://*](https://github.com/keras-team/keras) [*github.com/keras-team/keras*](https://github.com/keras-team/keras)). It was developed by François Chollet as part of aresearch project[12](#page18) and released as an open source project in March 2015. It quickly gained popularity owing to its ease-of-use, flexibility and beautiful design. To per‐ form the heavy computations required by neural networks, keras-team relies on a computation backend. At the present, you can choose from three popular open source deep learning libraries: TensorFlow, Microsoft Cognitive Toolkit (CNTK) or Theano.

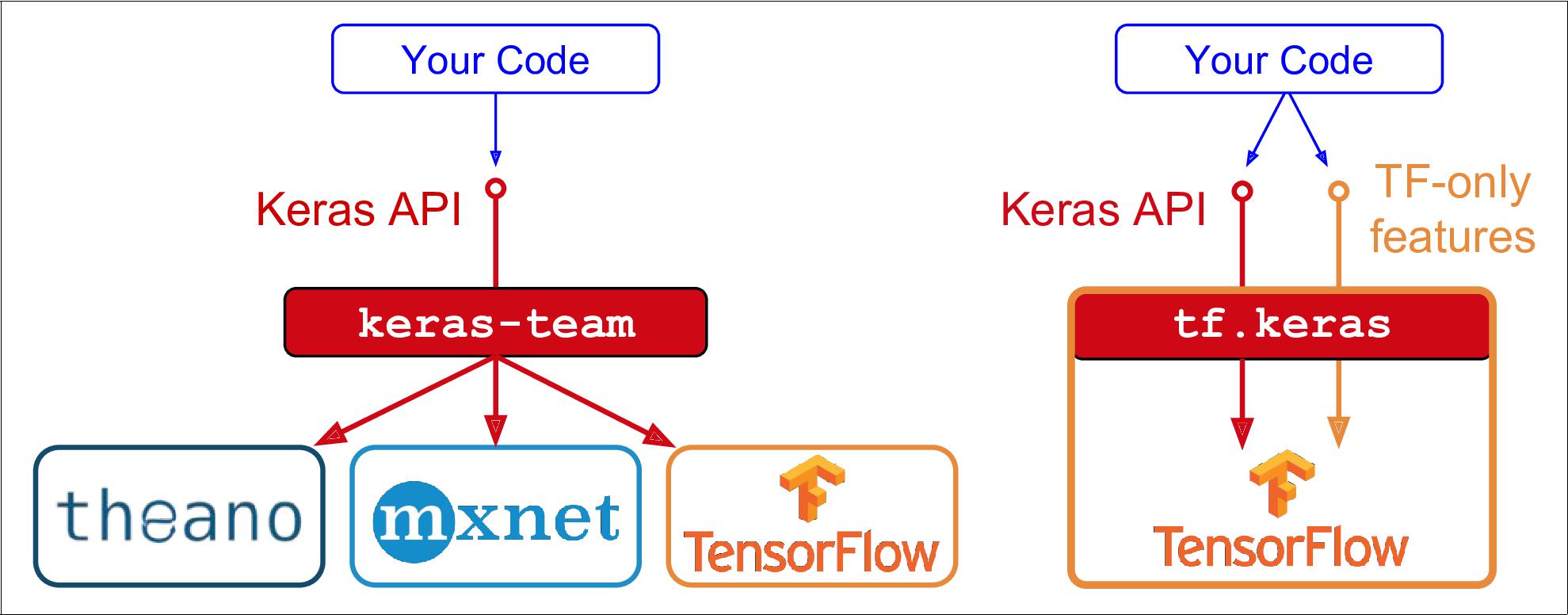
Moreover, since late 2016, other implementations have been released. You can now run Keras on Apache MXNet, Apple’s Core ML, Javascript or Typescript (to run Keras code in a web browser), or PlaidML (which can run on all sorts of GPU devices, not just Nvidia). Moreover, TensorFlow itself now comes bundled with its own Keras implementation called tf.keras. It only supports TensorFlow as the backend, but it has the advantage of offering some very useful extra features (see [Figure 10-10](#page19)): for example, it supports TensorFlow’s Data API which makes it quite easy to load and preprocess data efficiently. For this reason, we will use tf.keras in this book. However, in this chapter we will not use any of the TensorFlow-specific features, so the code should run fine on other Keras implementations as well (at least in Python), with only minor modifications, such as changing the imports.



12 Project ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System).



**292** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**



*Figure 10-10. Two Keras implementations: keras-team (le•) and tf.keras (right)*

As tf.keras is bundled with TensorFlow, let’s install TensorFlow!

**Installing TensorFlow 2**

Assuming you installed Jupyter and Scikit-Learn by following the installation instruc‐ tions in Chapter 2, you can simply use pip to install TensorFlow. If you created an isolated environment using virtualenv, you first need to activate it:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| $ cd $ML\_PATH | | # Your ML working directory (e.g., $HOME/ml) | | |
| $ | source env/bin/activate | # | on | Linux or MacOSX |
| $ | .\env\Scripts\activate | # | on | Windows |

Next, install TensorFlow 2 (if you are not using a virtualenv, you will need adminis‐ trator rights, or to add the --user option):

$ python3 -m pip install --upgrade tensorflow

For GPU support, you need to install tensorflow-gpu instead of tensorflow, and there are other libraries to install. See [*https://*](https://tensorflow.org/install/gpu) [*tensorflow.org/install/gpu*](https://tensorflow.org/install/gpu) formore details.



To test your installation, open a Python shell or a Jupyter notebook, then import Ten‐ sorFlow and tf.keras, and print their versions:

* **import tensorflow as tf**
* **from tensorflow import** keras
* tf.\_\_version\_\_

'2.0.0'

**>>>** keras.\_\_version\_\_'2.2.4-tf'



**Implementing MLPs with Keras** **|** **293**

The second version is the version of the Keras API implemented by tf.keras. Note that it ends with -tf, highlighting the fact that tf.keras implements the Keras API, plus some extra TensorFlow-specific features.

Now let’s use tf.keras! Let’s start by building a simple image classifier.

**Building an Image Classi€er Using the Sequential API**

First, we need to load a dataset. We will tackle *Fashion MNIST*, which is a drop-in replacement of MNIST (introduced in Chapter 3). It has the exact same format as MNIST (70,000 grayscale images of 28×28 pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is more diverse and the problem turns out to be significantly more challenging than MNIST. For example, a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST.

**Using Keras to Load the Dataset**

Keras provides some utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, the original California housing dataset, and more. Let’s load Fashion MNIST:

fashion\_mnist = keras.datasets.fashion\_mnist

(X\_train\_full, y\_train\_full), (X\_test, y\_test) = fashion\_mnist.load\_data()

When loading MNIST or Fashion MNIST using Keras rather than Scikit-Learn, one important difference is that every image is represented as a 28×28 array rather than a 1D array of size 784. Moreover, the pixel intensities are represented as integers (from 0 to 255) rather than floats (from 0.0 to 255.0). Here is the shape and data type of the training set:

* X\_train\_full.shape (60000, 28, 28)
* X\_train\_full.dtype dtype('uint8')

Note that the dataset is already split into a training set and a test set, but there is no validation set, so let’s create one. Moreover, since we are going to train the neural net‐ work using Gradient Descent, we must scale the input features. For simplicity, we just scale the pixel intensities down to the 0-1 range by dividing them by 255.0 (this also converts them to floats):

X\_valid, X\_train = X\_train\_full[:5000] / 255.0, X\_train\_full[5000:] / 255.0 y\_valid, y\_train = y\_train\_full[:5000], y\_train\_full[5000:]

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. However, for Fashion MNIST, we need the list of class names to know what we are dealing with:



**294** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

class\_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]

For example, the first image in the training set represents a coat:

**>>>** class\_names[y\_train[0]]'Coat'

[Figure 10-11](#page21) shows a few samples from the Fashion MNIST dataset:



*Figure 10-11. Samples from Fashion MNIST*

**Creating the Model Using the Sequential API**

Now let’s build the neural network! Here is a classification MLP with two hidden lay‐ ers:

model = keras.models.Sequential() model.add(keras.layers.Flatten(input\_shape=[28, 28])) model.add(keras.layers.Dense(300, activation="relu")) model.add(keras.layers.Dense(100, activation="relu")) model.add(keras.layers.Dense(10, activation="softmax"))

Let’s go through this code line by line:

* The first line creates a Sequential model. This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, con‐ nected sequentially. This is called the sequential API.
* Next, we build the first layer and add it to the model. It is a Flatten layer whose role is simply to convert each input image into a 1D array: if it receives input data X, it computes X.reshape(-1, 1). This layer does not have any parameters, it is just there to do some simple preprocessing. Since it is the first layer in the model, you should specify the input\_shape: this does not include the batch size, only the shape of the instances. Alternatively, you could add a keras.layers.InputLayer as the first layer, setting shape=[28,28].
* Next we add a Dense hidden layer with 300 neurons. It will use the ReLU activa‐ tion function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vec‐



**Implementing MLPs with Keras** **|** **295**

tor of bias terms (one per neuron). When it receives some input data, it computes [Equation 10-2](#page9).

* Next we add a second Dense hidden layer with 100 neurons, also using the ReLU activation function.
* Finally, we add a Dense output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).



Specifying activation="relu" is equivalent to activa tion=keras.activations.relu. Other activation functions are available in the keras.activations package, we will use many of them in this book. See [*https://keras.io/activations/*](https://keras.io/activations/) for the full list.

Instead of adding the layers one by one as we just did, you can pass a list of layers when creating the Sequential model:

model = keras.models.Sequential([ keras.layers.Flatten(input\_shape=[28, 28]), keras.layers.Dense(300, activation="relu"), keras.layers.Dense(100, activation="relu"), keras.layers.Dense(10, activation="softmax")

])



**Using Code Examples From keras.io**

Code examples documented on keras.io will work fine with tf.keras, but you need to change the imports. For example, consider this keras.io code:

**from keras.layers import** Denseoutput\_layer = Dense(10)

You must change the imports like this:

**from tensorflow.keras.layers import** Denseoutput\_layer = Dense(10)

Or simply use full paths, if you prefer:

**from tensorflow import** kerasoutput\_layer = keras.layers.Dense(10)

This is more verbose, but I use this approach in this book so you can easily see which packages to use, and to avoid confusion between standard classes and custom classes.

In production code, I use the previous approach, as do most people.



**296** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

The model’s summary() method displays all the model’s layers[13](#page23), including each layer’s name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parame‐ ters. The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters (we will see exam‐ ples of non-trainable parameters in Chapter 11):

**>>>** model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

flatten\_1 (Flatten) (None, 784) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_3 (Dense) (None, 300) 235500

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_4 (Dense) (None, 100) 30100

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_5 (Dense) (None, 10) 1010

=================================================================

Total params: 266,610 Trainable params: 266,610 Non-trainable params: 0

Note that Dense layers often have a *lot* of parameters. For example, the first hidden layer has 784 × 300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! This gives the model quite a lot of flexibility to fit the training data, but it also means that the model runs the risk of overfitting, especially when you do not have a lot of training data. We will come back to this later.

You can easily get a model’s list of layers, to fetch a layer by its index, or you can fetch it by name:

* model.layers [<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>, <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>, <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>, <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
* model.layers[1].name

'dense\_3'

**>>>** model.get\_layer('dense\_3').name'dense\_3'

All the parameters of a layer can be accessed using its get\_weights() and set\_weights() method. For a Dense layer, this includes both the connection weights and the bias terms:



13 You can also generate an image of your model using keras.utils.plot\_model().



**Implementing MLPs with Keras** **|** **297**

* weights, biases = hidden1.get\_weights()
* weights

array([[ 0.03854964, -0.04054524, 0.00599282, ..., 0.02566582, 0.01032123, 0.06914985],

..., [ 0.02632413, -0.05105981, -0.00332005, ..., 0.04175945,

0.0443138 , -0.05558084]], dtype=float32)

* weights.shape (784, 300)
* biases

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

**>>>** biases.shape(300,)

Notice that the Dense layer initialized the connection weights randomly (which is needed to break symmetry, as we discussed earlier), and the biases were just initial‐ ized to zeros, which is fine. If you ever want to use a different initialization method, you can set kernel\_initializer (*kernel* is another name for the matrix of connec‐ tion weights) or bias\_initializer when creating the layer. We will discuss initializ‐ ers further in Chapter 11, but if you want the full list, see [*https://keras.io/initializers/*](https://keras.io/initializers/).

The shape of the weight matrix depends on the number of inputs. This is why it is recommended to specify the input\_shape when creating the first layer in a Sequential model. However, if you do not specify the input shape, it’s okay: Keras will simply wait until it knows the input shape before it actually builds the model. This will happen either when you feed it actual data (e.g., during training), or when you call its build() method. Until the model is really built, the layers will not have any weights, and you will not be able to do certain things (such as print the model summary or save the model), so if you know the input shape when creating the model, it is best to specify it.



**Compiling the Model**

After a model is created, you must call its compile() method to specify the loss func‐ tion and the optimizer to use. Optionally, you can also specify a list of extra metrics to compute during training and evaluation:

model.compile(loss="sparse\_categorical\_crossentropy", optimizer="sgd",

metrics=["accuracy"])



**298** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

Using loss="sparse\_categorical\_crossentropy" is equivalent to loss=keras.losses.sparse\_categorical\_crossentropy. Simi‐



larly, optimizer="sgd" is equivalent to optimizer=keras.optimiz ers.SGD() and metrics=["accuracy"] is equivalent to

metrics=[keras.metrics.sparse\_categorical\_accuracy] (when using this loss). We will use many other losses, optimizers and met‐ rics in this book, but for the full lists see [*https://keras.io/losses/*](https://keras.io/losses/), [*https://keras.io/optimizers/*](https://keras.io/optimizers/) and[*https://keras.io/metrics/*](https://keras.io/metrics/).

This requires some explanation. First, we use the "sparse\_categorical\_crossen tropy" loss because we have sparse labels (i.e., for each instance there is just a target class index, from 0 to 9 in this case), and the classes are exclusive. If instead we had one target probability per class for each instance (such as one-hot vectors, e.g. [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.] to represent class 3), then we would need to use the "categorical\_crossentropy" loss instead. If we were doing binary classi‐ fication (with one or more binary labels), then we would use the "sigmoid" (i.e., logistic) activation function in the output layer instead of the "softmax" activation function, and we would use the "binary\_crossentropy" loss.



If you want to convert sparse labels (i.e., class indices) to one-hot vector labels, you can use the keras.utils.to\_categorical() function. To go the other way round, you can just use the np.arg max() function with axis=1.

Secondly, regarding the optimizer, "sgd" simply means that we will train the model using simple Stochastic Gradient Descent. In other words, Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff + Gradient Descent). We will discuss more efficient optimizers in Chapter 11 (they improve the Gradient Descent part, not the autodiff).

Finally, since this is a classifier, it’s useful to measure its "accuracy" during training and evaluation.

**Training and Evaluating the Model**

Now the model is ready to be trained. For this we simply need to call its fit() method. We pass it the input features (X\_train) and the target classes (y\_train), as well as the number of epochs to train (or else it would default to just 1, which would definitely not be enough to converge to a good solution). We also pass a validation set (this is optional): Keras will measure the loss and the extra metrics on this set at the end of each epoch, which is very useful to see how well the model really performs: if the performance on the training set is much better than on the validation set, your



**Implementing MLPs with Keras** **|** **299**

model is probably overfitting the training set (or there is a bug, such as a data mis‐ match between the training set and the validation set):

**>>>** history=model.fit(X\_train,y\_train,epochs=30,

**...** validation\_data=(X\_valid, y\_valid))

**...**

Train on 55000 samples, validate on 5000 samples Epoch 1/30

55000/55000 [==========] - 3s 55us/sample - loss: 1.4948 - acc: 0.5757

- val\_loss: 1.0042 - val\_acc: 0.7166

Epoch 2/30

55000/55000 [==========] - 3s 55us/sample - loss: 0.8690 - acc: 0.7318

- val\_loss: 0.7549 - val\_acc: 0.7616

[...] Epoch 50/50

55000/55000 [==========] - 4s 72us/sample - loss: 0.3607 - acc: 0.8752

- val\_loss: 0.3706 - val\_acc: 0.8728

And that’s it! The neural network is trained. At each epoch during training, Keras dis‐ plays the number of instances processed so far (along with a progress bar), the mean training time per sample, the loss and accuracy (or any other extra metrics you asked for), both on the training set and the validation set. You can see that the training loss went down, which is a good sign, and the validation accuracy reached 87.28% after 50 epochs, not too far from the training accuracy, so there does not seem to be much overfitting going on.



Instead of passing a validation set using the validation\_data argument, you could instead set validation\_split to the ratio of the training set that you want Keras to use for validation (e.g., 0.1).

If the training set was very skewed, with some classes being overrepresented and oth‐ ers underrepresented, it would be useful to set the class\_weight argument when calling the fit() method, giving a larger weight to underrepresented classes, and a lower weight to overrepresented classes. These weights would be used by Keras when computing the loss. If you need per-instance weights instead, you can set the sam ple\_weight argument (it supersedes class\_weight). This could be useful for exam‐ ple if some instances were labeled by experts while others were labeled using a crowdsourcing platform: you might want to give more weight to the former. You can also provide sample weights (but not class weights) for the validation set by adding them as a third item in the validation\_data tuple.

The fit() method returns a History object containing the training parameters (his tory.params), the list of epochs it went through (history.epoch), and most impor‐ tantly a dictionary (history.history) containing the loss and extra metrics it measured at the end of each epoch on the training set and on the validation set (if



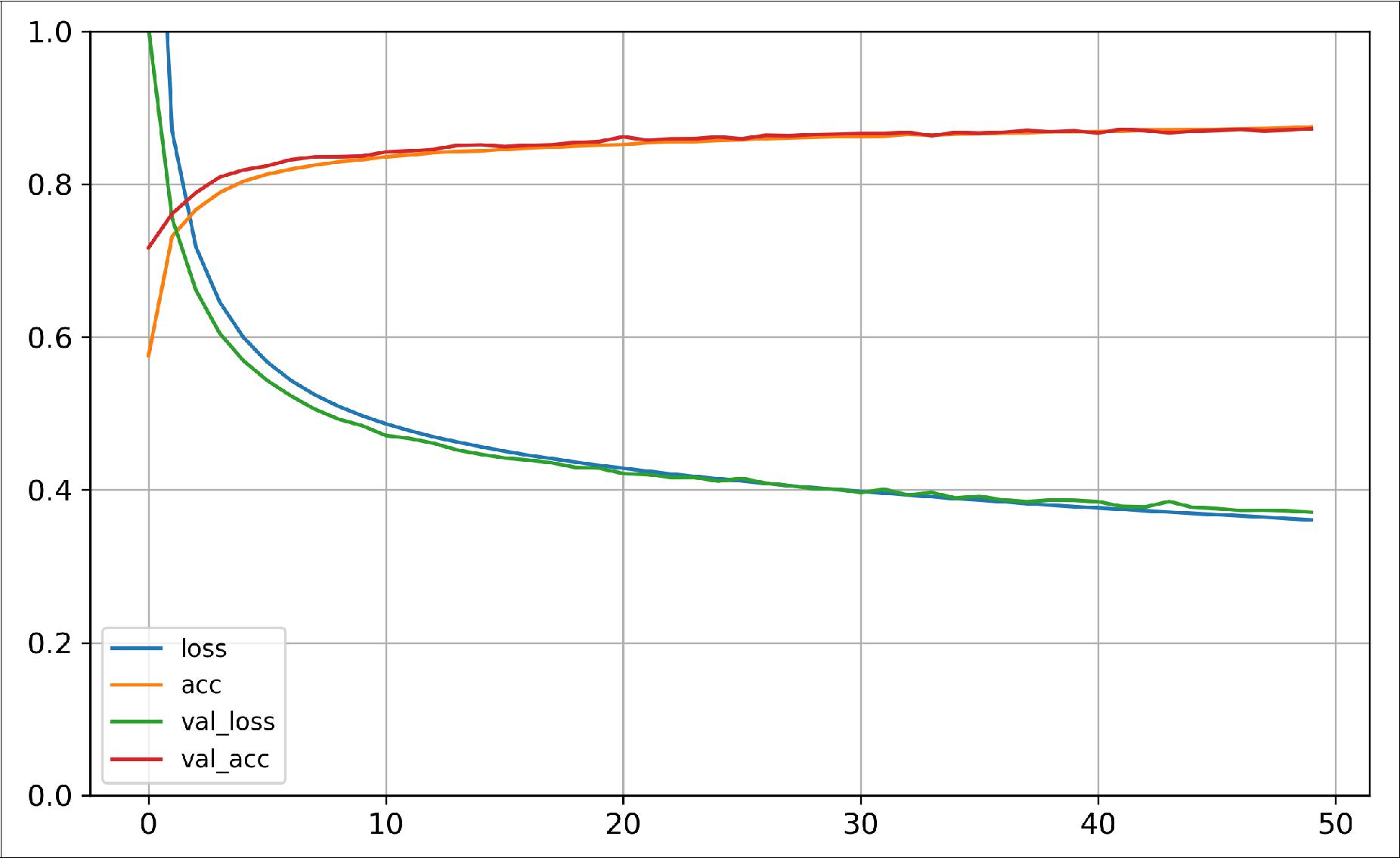
**300** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

any). If you create a Pandas DataFrame using this dictionary and call its plot() method, you get the learning curves shown in [Figure 10-12](#page27):

**import pandas as pd**

pd.DataFrame(history.history).plot(figsize=(8, 5)) plt.grid(True)

plt.gca().set\_ylim(0, 1) *# set the vertical range to [0-1]* plt.show()



*Figure 10-12. Learning Curves*

You can see that both the training and validation accuracy steadily increase during training, while the training and validation loss decrease. Good! Moreover, the valida‐ tion curves are quite close to the training curves, which means that there is not too much overfitting. In this particular case, the model performed better on the valida‐ tion set than on the training set at the beginning of training: this sometimes happens by chance (especially when the validation set is fairly small). However, the training set performance ends up beating the validation performance, as is generally the case when you train for long enough. You can tell that the model has not quite converged yet, as the validation loss is still going down, so you should probably continue train‐ ing. It’s as simple as calling the fit() method again, since Keras just continues train‐ ing where it left off (you should be able to reach close to 89% validation accuracy).

If you are not satisfied with the performance of your model, you should go back and tune the model’s hyperparameters, for example the number of layers, the number of neurons per layer, the types of activation functions we use for each hidden layer, the



**Implementing MLPs with Keras** **|** **301**

number of training epochs, the batch size (it can be set in the fit() method using the batch\_size argument, which defaults to 32). We will get back to hyperparameter tuning at the end of this chapter. Once you are satisfied with your model’s validation accuracy, you should evaluate it on the test set to estimate the generalization error before you deploy the model to production. You can easily do this using the evalu ate() method (it also supports several other arguments, such as batch\_size or sam ple\_weight, please check the documentation for more details):

**>>>** model.evaluate(X\_test,y\_test)

8832/10000 [==========================] - ETA: 0s - loss: 0.4074 - acc: 0.8540 [0.40738476498126985, 0.854]

As we saw in Chapter 2, it is common to get slightly lower performance on the test set than on the validation set, because the hyperparameters are tuned on the validation set, not the test set (however, in this example, we did not do any hyperparameter tun‐ ing, so the lower accuracy is just bad luck). Remember to resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

**Using the Model to Make Predictions**

Next, we can use the model’s predict() method to make predictions on new instan‐ ces. Since we don’t have actual new instances, we will just use the first 3 instances of the test set:

* X\_new = X\_test[:3]
* y\_proba = model.predict(X\_new)
* y\_proba.round(2)

array([[0. , 0. , 0. , 0. , 0. , 0.09, 0. , 0.12, 0. , 0.79], [0. , 0. , 0.94, 0. , 0.02, 0. , 0.04, 0. , 0. , 0. ], [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]], dtype=float32)

As you can see, for each instance the model estimates one probability per class, from class 0 to class 9. For example, for the first image it estimates that the probability of class 9 (ankle boot) is 79%, the probability of class 7 (sneaker) is 12%, the probability of class 5 (sandal) is 9%, and the other classes are negligible. In other words, it “believes” it’s footwear, probably ankle boots, but it’s not entirely sure, it might be sneakers or sandals instead. If you only care about the class with the highest estima‐ ted probability (even if that probability is quite low) then you can use the pre dict\_classes() method instead:

* y\_pred = model.predict\_classes(X\_new)
* y\_pred

array([9, 2, 1])

**>>>** np.array(class\_names)[y\_pred]

array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')

And the classifier actually classified all three images correctly:



**302** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

* y\_new = y\_test[:3]
* y\_new

array([9, 2, 1])

Now you know how to build, train, evaluate and use a classification MLP using the Sequential API. But what about regression?

**Building a Regression MLP Using the Sequential API**

Let’s switch to the California housing problem and tackle it using a regression neural network. For simplicity, we will use Scikit-Learn’s fetch\_california\_housing() function to load the data: this dataset is simpler than the one we used in Chapter 2, since it contains only numerical features (there is no ocean\_proximity feature), and there is no missing value. After loading the data, we split it into a training set, a vali‐ dation set and a test set, and we scale all the features:

**from sklearn.datasets import** fetch\_california\_housing **from sklearn.model\_selection import** train\_test\_split **from sklearn.preprocessing import** StandardScaler

housing = fetch\_california\_housing()

X\_train\_full, X\_test, y\_train\_full, y\_test = train\_test\_split( housing.data, housing.target)

X\_train, X\_valid, y\_train, y\_valid = train\_test\_split( X\_train\_full, y\_train\_full)

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train) X\_valid\_scaled = scaler.transform(X\_valid) X\_test\_scaled = scaler.transform(X\_test)

Building, training, evaluating and using a regression MLP using the Sequential API to make predictions is quite similar to what we did for classification. The main differ‐ ences are the fact that the output layer has a single neuron (since we only want to predict a single value) and uses no activation function, and the loss function is the mean squared error. Since the dataset is quite noisy, we just use a single hidden layer with fewer neurons than before, to avoid overfitting:

model = keras.models.Sequential([

keras.layers.Dense(30, activation="relu", input\_shape=X\_train.shape[1:]), keras.layers.Dense(1)

])

model.compile(loss="mean\_squared\_error", optimizer="sgd") history = model.fit(X\_train, y\_train, epochs=20,

validation\_data=(X\_valid, y\_valid)) mse\_test = model.evaluate(X\_test, y\_test)

X\_new = X\_test[:3] *# pretend these are new instances* y\_pred = model.predict(X\_new)



**Implementing MLPs with Keras** **|** **303**

As you can see, the Sequential API is quite easy to use. However, although sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the Functional API.

**Building Complex Models Using the Functional API**

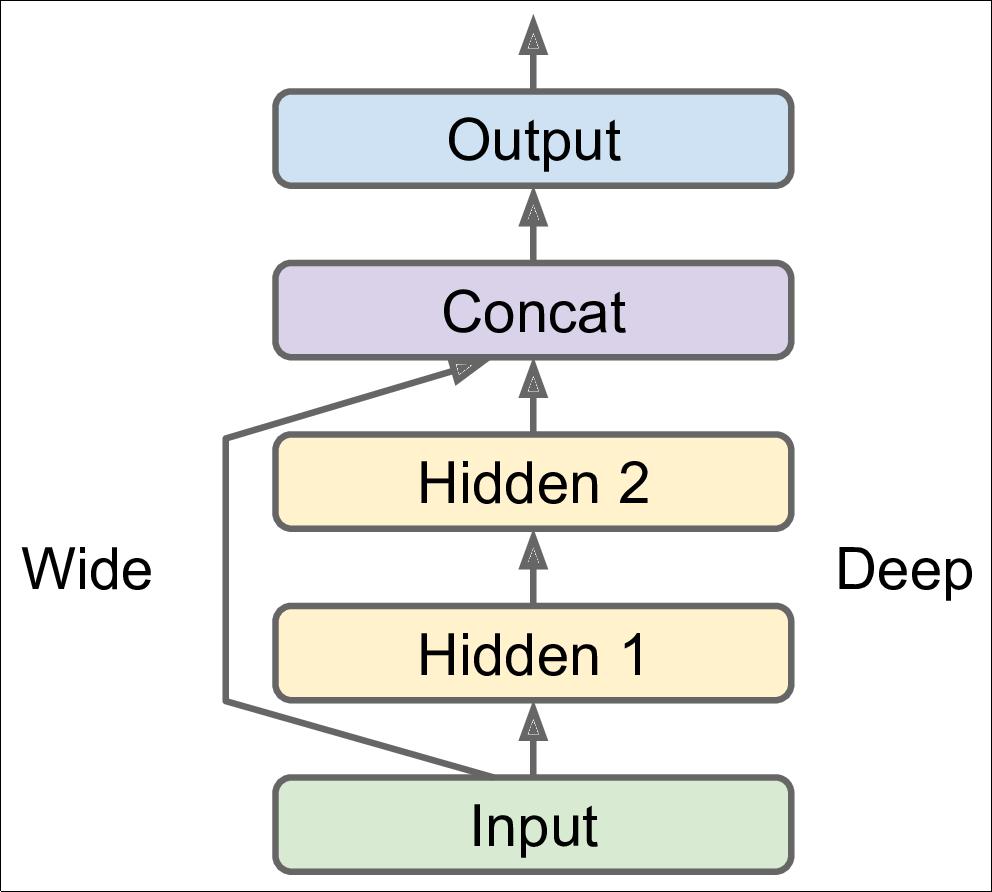
One example of a non-sequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a [2016 paper](https://arxiv.org/abs/1606.07792) by Heng-Tze Cheng et al.[14](#page30). It connects all or part of the inputs directly to the output layer, as shown in [Figure 10-13](#page31). This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). In contrast, a regular MLP forces all the data to flow through the full stack of layers, thus simple patterns in the data may end up being distorted by this sequence of transfor‐ mations.



14 “Wide & Deep Learning for Recommender Systems,” Heng-Tze Cheng et al. (2016).



**304** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**



*Figure 10-13. Wide and Deep Neural Network*

Let’s build such a neural network to tackle the California housing problem:

input = keras.layers.Input(shape=X\_train.shape[1:]) hidden1 = keras.layers.Dense(30, activation="relu")(input)

hidden2 = keras.layers.Dense(30, activation="relu")(hidden1) concat = keras.layers.Concatenate()[input, hidden2])

output = keras.layers.Dense(1)(concat)

model = keras.models.Model(inputs=[input], outputs=[output])

Let’s go through each line of this code:

* First, we need to create an Input object. This is needed because we may have multiple inputs, as we will see later.
* Next, we create a Dense layer with 30 neurons and using the ReLU activation function. As soon as it is created, notice that we call it like a function, passing it the input. This is why this is called the Functional API. Note that we are just tell‐



**Implementing MLPs with Keras** **|** **305**

ing Keras how it should connect the layers together, no actual data is being pro‐ cessed yet.

* We then create a second hidden layer, and again we use it as a function. Note however that we pass it the output of the first hidden layer.
* Next, we create a Concatenate() layer, and once again we immediately use it like a function, to concatenate the input and the output of the second hidden layer (you may prefer the keras.layers.concatenate() function, which creates a Con catenate layer and immediately calls it with the given inputs).
* Then we create the output layer, with a single neuron and no activation function, and we call it like a function, passing it the result of the concatenation.
* Lastly, we create a Keras Model, specifying which inputs and outputs to use.

Once you have built the Keras model, everything is exactly like earlier, so no need to repeat it here: you must compile the model, train it, evaluate it and use it to make predictions.

But what if you want to send a subset of the features through the wide path, and a different subset (possibly overlapping) through the deep path (see [Figure 10-14](#page33))? In this case, one solution is to use multiple inputs. For example, suppose we want to send 5 features through the deep path (features 0 to 4), and 6 features through the wide path (features 2 to 7):

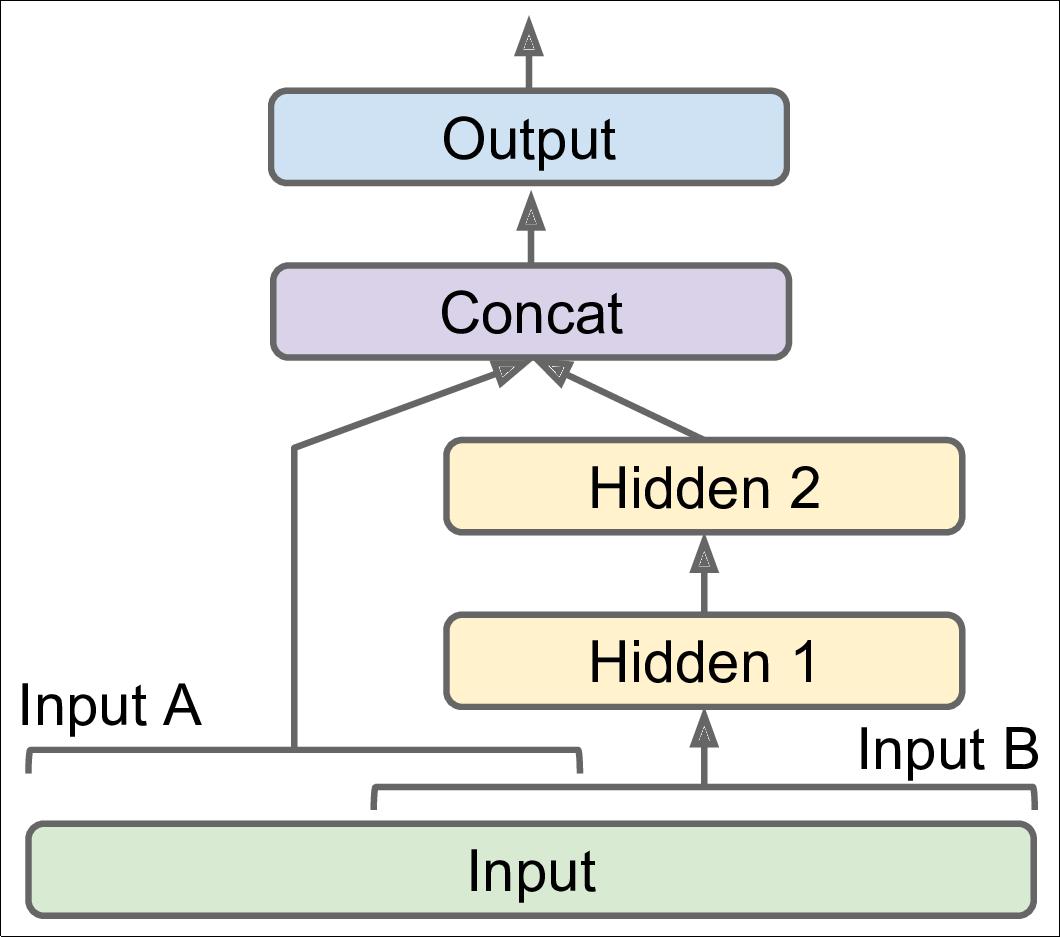
input\_A = keras.layers.Input(shape=[5]) input\_B = keras.layers.Input(shape=[6])

hidden1 = keras.layers.Dense(30, activation="relu")(input\_B) hidden2 = keras.layers.Dense(30, activation="relu")(hidden1) concat = keras.layers.concatenate([input\_A, hidden2]) output = keras.layers.Dense(1)(concat)

model = keras.models.Model(inputs=[input\_A, input\_B], outputs=[output])



**306** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**



*Figure 10-14. Handling Multiple Inputs*

The code is self-explanatory. Note that we specified inputs=[input\_A, input\_B] when creating the model. Now we can compile the model as usual, but when we call the fit() method, instead of passing a single input matrix X\_train, we must pass a pair of matrices (X\_train\_A, X\_train\_B): one per input. The same is true for

X\_valid, and also for X\_test and X\_new when you call evaluate() or predict():

model.compile(loss="mse", optimizer="sgd")

X\_train\_A, X\_train\_B = X\_train[:, :5], X\_train[:, 2:]

X\_valid\_A, X\_valid\_B = X\_valid[:, :5], X\_valid[:, 2:]

X\_test\_A, X\_test\_B = X\_test[:, :5], X\_test[:, 2:]

X\_new\_A, X\_new\_B = X\_test\_A[:3], X\_test\_B[:3]

history = model.fit((X\_train\_A, X\_train\_B), y\_train, epochs=20, validation\_data=((X\_valid\_A, X\_valid\_B), y\_valid))

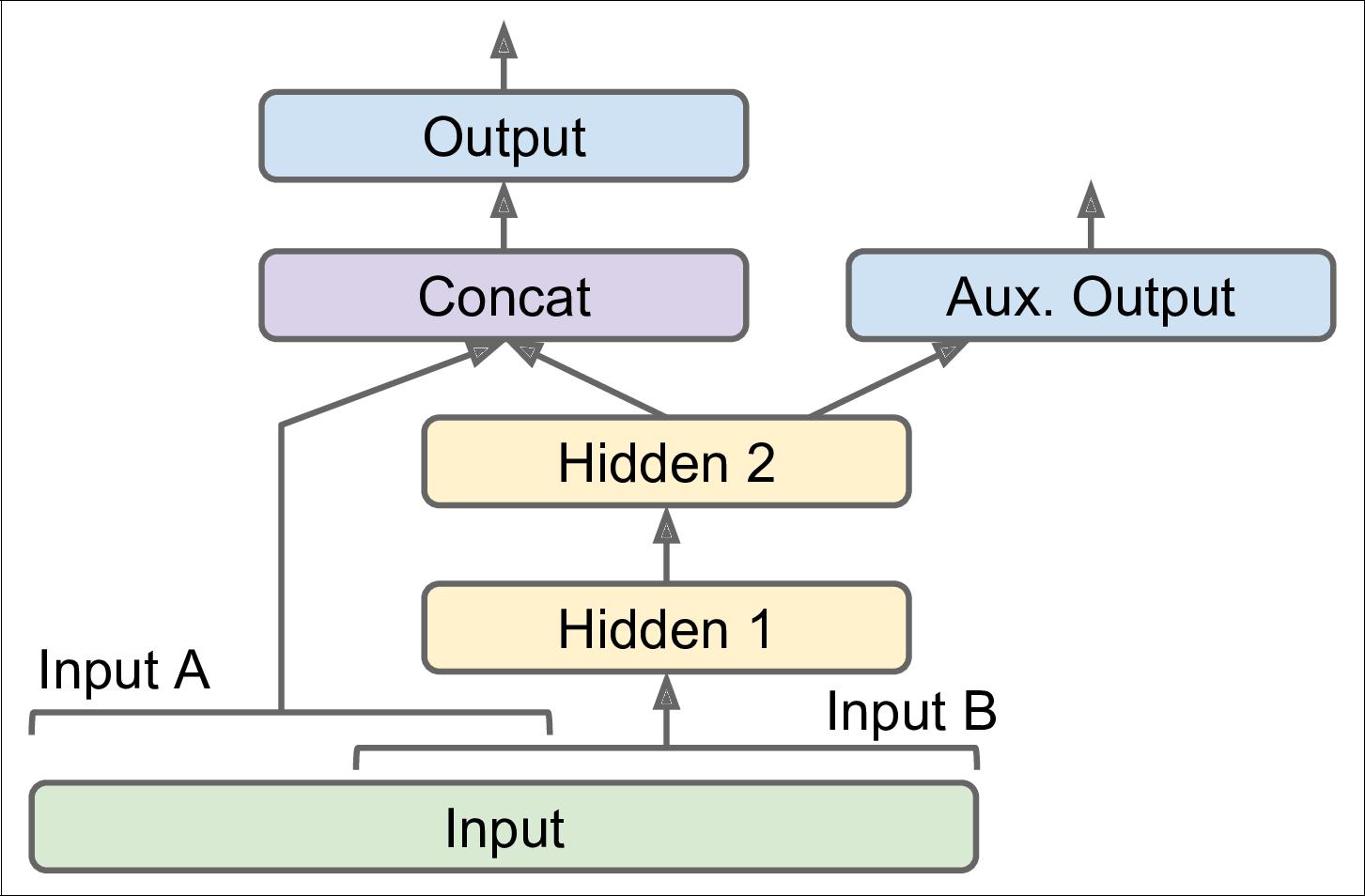
mse\_test = model.evaluate((X\_test\_A, X\_test\_B), y\_test) y\_pred = model.predict((X\_new\_A, X\_new\_B))



**Implementing MLPs with Keras** **|** **307**

There are also many use cases in which you may want to have multiple outputs:

* The task may demand it, for example you may want to locate and classify the main object in a picture. This is both a regression task (finding the coordinates of the object’s center, as well as its width and height) and a classification task.
* Similarly, you may have multiple independent tasks to perform based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks.
* Another use case is as a regularization technique (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model’s ability to general‐ ize). For example, you may want to add some auxiliary outputs in a neural net‐ work architecture (see [Figure 10-15](#page34)) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.



*Figure 10-15. Handling Multiple Outputs – Auxiliary Output for Regularization*

Adding extra outputs is quite easy: just connect them to the appropriate layers and add them to your model’s list of outputs. For example, the following code builds the network represented in [Figure 10-15](#page34):



**308** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

[...] *# Same as above, up to the main output layer* output = keras.layers.Dense(1)(concat)

aux\_output = keras.layers.Dense(1)(hidden2)

model = keras.models.Model(inputs=[input\_A, input\_B], outputs=[output, aux\_output])

Each output will need its own loss function, so when we compile the model we should pass a list of losses (if we pass a single loss, Keras will assume that the same loss must be used for all outputs). By default, Keras will compute all these losses and simply add them up to get the final loss used for training. However, we care much more about the main output than about the auxiliary output (as it is just used for reg‐ ularization), so we want to give the main output’s loss a much greater weight. Fortu‐ nately, it is possible to set all the loss weights when compiling the model:

model.compile(loss=["mse", "mse"], loss\_weights=[0.9, 0.1], optimizer="sgd")

Now when we train the model, we need to provide some labels for each output. In this example, the main output and the auxiliary output should try to predict the same thing, so they should use the same labels. So instead of passing y\_train, we just need to pass (y\_train, y\_train) (and the same goes for y\_valid and y\_test):

history = model.fit(

[X\_train\_A, X\_train\_B], [y\_train, y\_train], epochs=20, validation\_data=([X\_valid\_A, X\_valid\_B], [y\_valid, y\_valid]))

When we evaluate the model, Keras will return the total loss, as well as all the individ‐ ual losses:

total\_loss, main\_loss, aux\_loss = model.evaluate( [X\_test\_A, X\_test\_B], [y\_test, y\_test])

Similarly, the predict() method will return predictions for each output:

y\_pred\_main, y\_pred\_aux = model.predict([X\_new\_A, X\_new\_B])

As you can see, you can build any sort of architecture you want quite easily with the Functional API. Let’s look at one last way you can build Keras models.

**Building Dynamic Models Using the Subclassing API**

Both the Sequential API and the Functional API are declarative: you start by declar‐ ing which layers you want to use and how they should be connected, and only then can you start feeding the model some data for training or inference. This has many advantages: the model can easily be saved, cloned, shared, its structure can be dis‐ played and analyzed, the framework can infer shapes and check types, so errors can be caught early (i.e., before any data ever goes through the model). It’s also fairly easy to debug, since the whole model is just a static graph of layers. But the flip side is just that: it’s static. Some models involve loops, varying shapes, conditional branching, and other dynamic behaviors. For such cases, or simply if you prefer a more impera‐ tive programming style, the Subclassing API is for you.



**Implementing MLPs with Keras** **|** **309**

Simply subclass the Model class, create the layers you need in the constructor, and use them to perform the computations you want in the call() method. For example, cre‐ ating an instance of the following WideAndDeepModel class gives us an equivalent model to the one we just built with the Functional API. You can then compile it, eval‐ uate it and use it to make predictions, exactly like we just did.

**class WideAndDeepModel**(keras.models.Model):

**def** **\_\_init\_\_**(self,units=30,activation="relu",\*\*kwargs):super().**\_\_init\_\_**(\*\*kwargs) *# handles standard args (e.g., name)* self.hidden1 = keras.layers.Dense(units, activation=activation) self.hidden2 = keras.layers.Dense(units, activation=activation) self.main\_output = keras.layers.Dense(1)

self.aux\_output = keras.layers.Dense(1)

**def** call(self, inputs): input\_A, input\_B = inputs

hidden1 = self.hidden1(input\_B) hidden2 = self.hidden2(hidden1)

concat = keras.layers.concatenate([input\_A, hidden2]) main\_output = self.main\_output(concat)

aux\_output = self.aux\_output(hidden2) **return** main\_output,aux\_output

model = WideAndDeepModel()

This example looks very much like the Functional API, except we do not need to cre‐ ate the inputs, we just use the input argument to the call() method, and we separate the creation of the layers[15](#page36) in the constructor from their usage in the call() method. However, the big difference is that you can do pretty much anything you want in the call() method: for loops, if statements, low-level TensorFlow operations, your imagination is the limit (see Chapter 12)! This makes it a great API for researchers experimenting with new ideas.

However, this extra flexibility comes at a cost: your model’s architecture is hidden within the call() method, so Keras cannot easily inspect it, it cannot save or clone it, and when you call the summary() method, you only get a list of layers, without any information on how they are connected to each other. Moreover, Keras cannot check types and shapes ahead of time, and it is easier to make mistakes. So unless you really need that extra flexibility, you should probably stick to the Sequential API or the Functional API.



1. Keras models have an output attribute, so we cannot use that name for the main output layer, which is why we renamed it to main\_output.



**310** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

Keras models can be used just like regular layers, so you can easily compose them to build complex architectures.

Now that you know how to build and train neural nets using Keras, you will want to save them!

**Saving and Restoring a Model**

Saving a trained Keras model is as simple as it gets:

model.save("my\_keras\_model.h5")

Keras will save both the model’s architecture (including every layer’s hyperparame‐ ters) and the value of all the model parameters for every layer (e.g., connection weights and biases), using the HDF5 format. It also saves the optimizer (including its hyperparameters and any state it may have).

You will typically have a script that trains a model and saves it, and one or more scripts (or web services) that load the model and use it to make predictions. Loading the model is just as easy:

model = keras.models.load\_model("my\_keras\_model.h5")

This will work when using the Sequential API or the Functional API, but unfortunately not when using Model subclassing. How‐ ever, you can use save\_weights() and load\_weights() to at least save and restore the model parameters (but you will need to save and restore everything else yourself).



But what if training lasts several hours? This is quite common, especially when train‐ ing on large datasets. In this case, you should not only save your model at the end of training, but also save checkpoints at regular intervals during training. But how can you tell the fit() method to save checkpoints? The answer is: using callbacks.

**Using Callbacks**

The fit() method accepts a callbacks argument that lets you specify a list of objects that Keras will call during training at the start and end of training, at the start and end of each epoch and even before and after processing each batch. For example, the Mod elCheckpoint callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:



**Implementing MLPs with Keras** **|** **311**

[...] *# build and compile the model*

checkpoint\_cb = keras.callbacks.ModelCheckpoint("my\_keras\_model.h5") history = model.fit(X\_train, y\_train, epochs=10, callbacks=[checkpoint\_cb])

Moreover, if you use a validation set during training, you can set save\_best\_only=True when creating the ModelCheckpoint. In this case, it will only save your model when its performance on the validation set is the best so far. This way, you do not need to worry about training for too long and overfitting the training set: simply restore the last model saved after training, and this will be the best model on the validation set. This is a simple way to implement early stopping (introduced in Chapter 4):

checkpoint\_cb = keras.callbacks.ModelCheckpoint("my\_keras\_model.h5", save\_best\_only=True)

history = model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_valid, y\_valid), callbacks=[checkpoint\_cb])

model = keras.models.load\_model("my\_keras\_model.h5") *# rollback to best model*

Another way to implement early stopping is to simply use the EarlyStopping call‐ back. It will interrupt training when it measures no progress on the validation set for a number of epochs (defined by the patience argument), and it will optionally roll back to the best model. You can combine both callbacks to both save checkpoints of your model (in case your computer crashes), and actually interrupt training early when there is no more progress (to avoid wasting time and resources):

early\_stopping\_cb = keras.callbacks.EarlyStopping(patience=10, restore\_best\_weights=True)

history = model.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_valid, y\_valid), callbacks=[checkpoint\_cb, early\_stopping\_cb])

The number of epochs can be set to a large value since training will stop automati‐ cally when there is no more progress. Moreover, there is no need to restore the best model saved in this case since the EarlyStopping callback will keep track of the best weights and restore them for us at the end of training.



There are many other callbacks available in the keras.callbacks package. See [*https://keras.io/callbacks/*](https://keras.io/callbacks/).

If you need extra control, you can easily write your own custom callbacks. For exam‐ ple, the following custom callback will display the ratio between the validation loss and the training loss during training (e.g., to detect overfitting):



**312** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

**class PrintValTrainRatioCallback**(keras.callbacks.Callback): **def** on\_epoch\_end(self,epoch,logs):

**print**("**\n**val/train: {:.2f}".format(logs["val\_loss"]/logs["loss"]))

As you might expect, you can implement on\_train\_begin(), on\_train\_end(),

on\_epoch\_begin(), on\_epoch\_begin(), on\_batch\_end() and on\_batch\_end(). Moreover, callbacks can also be used during evaluation and predictions, should you ever need them (e.g., for debugging). In this case, you should implement

on\_test\_begin(), on\_test\_end(), on\_test\_batch\_begin(), or on\_test\_batch\_end() (called by evaluate()), or on\_predict\_begin(), on\_pre dict\_end(), on\_predict\_batch\_begin(), or on\_predict\_batch\_end() (called by

predict()).

Now let’s take a look at one more tool you should definitely have in your toolbox when using tf.keras: TensorBoard.

**Visualization Using TensorBoard**

TensorBoard is a great interactive visualization tool that you can use to view the learning curves during training, compare learning curves between multiple runs, vis‐ ualize the computation graph, analyze training statistics, view images generated by your model, visualize complex multidimensional data projected down to 3D and automatically clustered for you, and more! This tool is installed automatically when you install TensorFlow, so you already have it!

To use it, you must modify your program so that it outputs the data you want to visu‐ alize to special binary log files called *event files*. Each binary data record is called a *summary*. The TensorBoard server will monitor the log directory, and it will automat‐ically pick up the changes and update the visualizations: this allows you to visualize live data (with a short delay), such as the learning curves during training. In general, you want to point the TensorBoard server to a root log directory, and configure your program so that it writes to a different subdirectory every time it runs. This way, the same TensorBoard server instance will allow you to visualize and compare data from multiple runs of your program, without getting everything mixed up.

So let’s start by defining the root log directory we will use for our TensorBoard logs, plus a small function that will generate a subdirectory path based on the current date and time, so that it is different at every run. You may want to include extra informa‐ tion in the log directory name, such as hyperparameter values that you are testing, to make it easier to know what you are looking at in TensorBoard:

root\_logdir = os.path.join(os.curdir, "my\_logs")

**def** get\_run\_logdir(): **import time**

run\_id = time.strftime("run\_%Y\_%m\_%d-%H\_%M\_%S") **return** os.path.join(root\_logdir,run\_id)



**Implementing MLPs with Keras** **|** **313**

run\_logdir = get\_run\_logdir() *# e.g., './my\_logs/run\_2019\_01\_16-11\_28\_43'*

Next, the good news is that Keras provides a nice TensorBoard callback:

[...] *# Build and compile your model*

tensorboard\_cb = keras.callbacks.TensorBoard(run\_logdir) history = model.fit(X\_train, y\_train, epochs=30,

validation\_data=(X\_valid, y\_valid), callbacks=[tensorboard\_cb])

And that’s all there is to it! It could hardly be easier to use. If you run this code, the TensorBoard callback will take care of creating the log directory for you (along with its parent directories if needed), and during training it will create event files and write summaries to them. After running the program a second time (perhaps changing some hyperparameter value), you will end up with a directory structure similar to this one:

my\_logs

├── run\_2019\_01\_16-16\_51\_02

│ └── events.out.tfevents.1547628669.mycomputer.local.v2 └── run\_2019\_01\_16-16\_56\_50

└── events.out.tfevents.1547629020.mycomputer.local.v2

Next you need to start the TensorBoard server. If you installed TensorFlow within a virtualenv, you should activate it. Next, run the following command at the root of the project (or from anywhere else as long as you point to the appropriate log directory). If your shell cannot find the tensorboard script, then you must update your PATH environment variable so that it contains the directory in which the script was installed (alternatively, you can just replace tensorboard with python3 -m tensor

board.main).

$ tensorboard --logdir=./my\_logs --port=6006

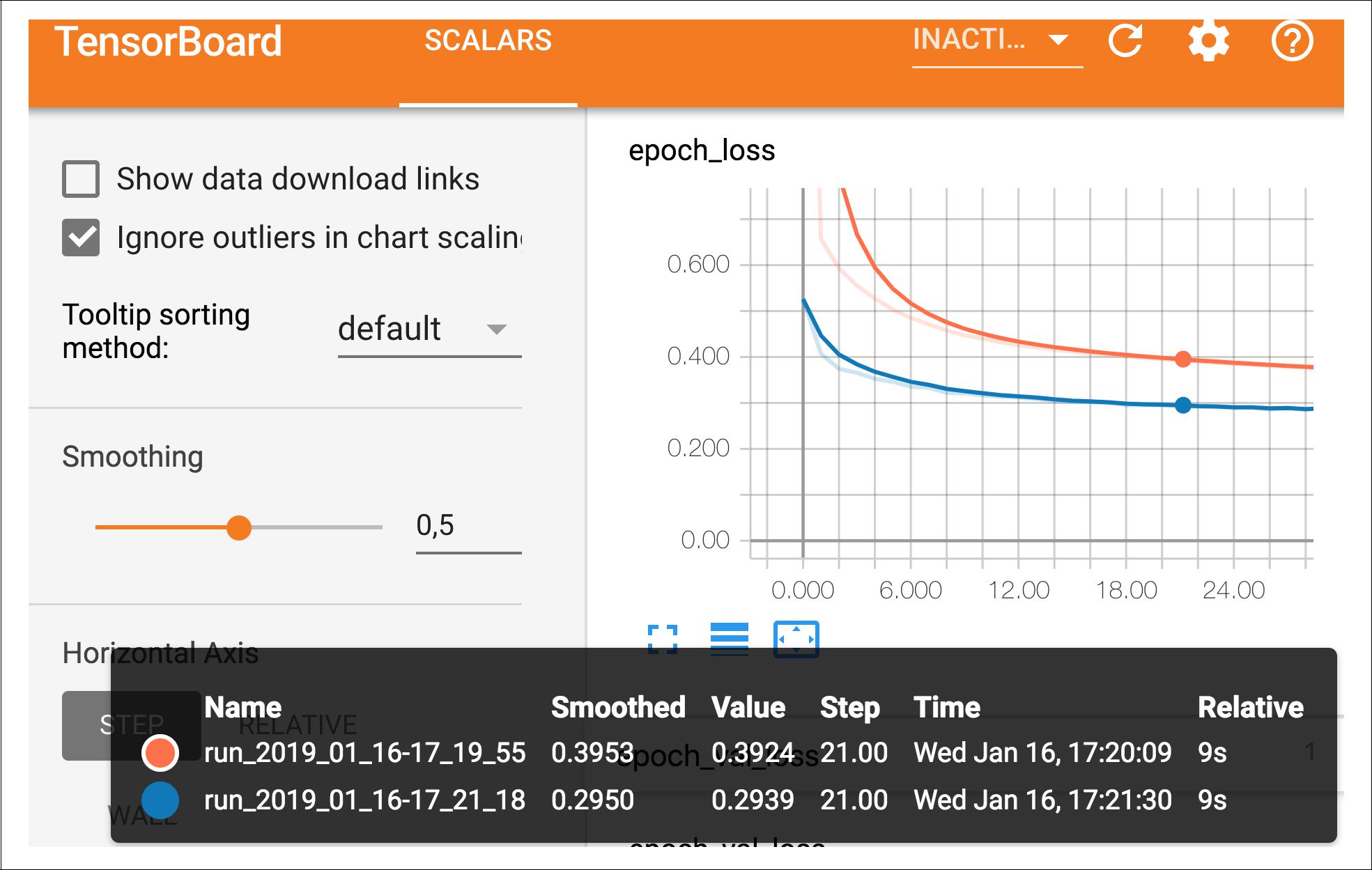
TensorBoard 2.0.0 at http://mycomputer.local:6006 (Press CTRL+C to quit)

Finally, open up a web browser to [*http://localhost:6006*](http://localhost:6006). You should see TensorBoard’s web interface. Click on the SCALARS tab to view the learning curves (see [Figure 10-16](#page41)). Notice that the training loss went down nicely during both runs, but the second run went down much faster. Indeed, we used a larger learning rate by set‐

ting optimizer=keras.optimizers.SGD(lr=0.05) instead of optimizer="sgd", which defaults to a learning rate of 0.001.



**314** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**



*Figure 10-16. Visualizing Learning Curves with TensorBoard*

Unfortunately, at the time of writing, no other data is exported by the TensorBoard callback, but this issue will probably be fixed by the time you read these lines. In Ten‐ sorFlow 1, this callback exported the computation graph and many useful statistics: type help(keras.callbacks.TensorBoard) to see all the options.

Let’s summarize what you learned so far in this chapter: we saw where neural nets came from, what an MLP is and how you can use it for classification and regression, how to build MLPs using tf.keras’s Sequential API, or more complex architectures using the Functional API or Model Subclassing, you learned how to save and restore a model, use callbacks for checkpointing, early stopping, and more, and finally how to use TensorBoard for visualization. You can already go ahead and use neural networks to tackle many problems! However, you may wonder how to choose the number of hidden layers, the number of neurons in the network, and all the other hyperparame‐ ters. Let’s look at this now.

**Fine-Tuning Neural Network Hyperparameters**

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network architec‐ ture, but even in a simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initi‐



**Fine-Tuning Neural Network Hyperparameters** **|** **315**

alization logic, and much more. How do you know what combination of hyperpara‐ meters is the best for your task?

One option is to simply try many combinations of hyperparameters and see which one works best on the validation set (or using K-fold cross-validation). For this, one approach is simply use GridSearchCV or RandomizedSearchCV to explore the hyper‐ parameter space, as we did in Chapter 2. For this, we need to wrap our Keras models in objects that mimic regular Scikit-Learn regressors. The first step is to create a func‐ tion that will build and compile a Keras model, given a set of hyperparameters:

**def** build\_model(n\_hidden=1, n\_neurons=30, learning\_rate=3e-3, input\_shape=[8]): model = keras.models.Sequential()

options = {"input\_shape": input\_shape} **for** layer **in** range(n\_hidden):

model.add(keras.layers.Dense(n\_neurons, activation="relu", \*\*options)) options = {}

model.add(keras.layers.Dense(1, \*\*options)) optimizer = keras.optimizers.SGD(learning\_rate) model.compile(loss="mse", optimizer=optimizer) **return** model

This function creates a simple Sequential model for univariate regression (only one output neuron), with the given input shape and the given number of hidden layers and neurons, and it compiles it using an SGD optimizer configured with the given learning rate. The options dict is used to ensure that the first layer is properly given the input shape (note that if n\_hidden=0, the first layer will be the output layer). It is good practice to provide reasonable defaults to as many hyperparameters as you can, as Scikit-Learn does.

Next, let’s create a KerasRegressor based on this build\_model() function:

keras\_reg = keras.wrappers.scikit\_learn.KerasRegressor(build\_model)

The KerasRegressor object is a thin wrapper around the Keras model built using build\_model(). Since we did not specify any hyperparameter when creating it, it will just use the default hyperparameters we defined in build\_model(). Now we can use this object like a regular Scikit-Learn regressor: we can train it using its fit() method, then evaluate it using its score() method, and use it to make predictions using its predict() method. Note that any extra parameter you pass to the fit() method will simply get passed to the underlying Keras model. Also note that the score will be the opposite of the MSE because Scikit-Learn wants scores, not losses (i.e., higher should be better).

keras\_reg.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_valid, y\_valid), callbacks=[keras.callbacks.EarlyStopping(patience=10)])

mse\_test = keras\_reg.score(X\_test, y\_test) y\_pred = keras\_reg.predict(X\_new)



**316** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

However, we do not actually want to train and evaluate a single model like this, we want to train hundreds of variants and see which one performs best on the validation set. Since there are many hyperparameters, it is preferable to use a randomized search rather than grid search (as we discussed in Chapter 2). Let’s try to explore the number of hidden layers, the number of neurons and the learning rate:

**from scipy.stats import** reciprocal

**from sklearn.model\_selection import** RandomizedSearchCV

param\_distribs = { "n\_hidden": [0, 1, 2, 3],

"n\_neurons": np.arange(1, 100), "learning\_rate": reciprocal(3e-4, 3e-2),

}

rnd\_search\_cv = RandomizedSearchCV(keras\_reg, param\_distribs, n\_iter=10, cv=3) rnd\_search\_cv.fit(X\_train, y\_train, epochs=100,

validation\_data=(X\_valid, y\_valid), callbacks=[keras.callbacks.EarlyStopping(patience=10)])

As you can see, this is identical to what we did in Chapter 2, with the exception that we pass extra parameters to the fit() method: they simply get relayed to the under‐ lying Keras models. Note that RandomizedSearchCV uses K-fold cross-validation, so it does not use X\_valid and y\_valid. These are just used for early stopping.

The exploration may last many hours depending on the hardware, the size of the dataset, the complexity of the model and the value of n\_iter and cv. When it is over, you can access the best parameters found, the best score, and the trained Keras model like this:

**>>>** rnd\_search\_cv.best\_params\_

{'learning\_rate': 0.0033625641252688094, 'n\_hidden': 2, 'n\_neurons': 42}

* rnd\_search\_cv.best\_score\_ -0.3189529188278931
* model = rnd\_search\_cv.best\_estimator\_.model

You can now save this model, evaluate it on the test set, and if you are satisfied with its performance, deploy it to production. Using randomized search is not too hard, and it works well for many fairly simple problems. However, when training is slow (e.g., for more complex problems with larger datasets), this approach will only explore a tiny portion of the hyperparameter space. You can partially alleviate this problem by assisting the search process manually: first run a quick random search using wide ranges of hyperparameter values, then run another search using smaller ranges of values centered on the best ones found during the first run, and so on. This will hopefully zoom in to a good set of hyperparameters. However, this is very time consuming, and probably not the best use of your time.

Fortunately, there are many techniques to explore a search space much more effi‐ ciently than randomly. Their core idea is simple: when a region of the space turns out



**Fine-Tuning Neural Network Hyperparameters** **|** **317**

to be good, it should be explored more. This takes care of the “zooming” process for you and leads to much better solutions in much less time. Here are a few Python libraries you can use to optimize hyperparameters:

* [Hyperopt](https://github.com/hyperopt/hyperopt): a popular Python library for optimizing over all sorts of complex search spaces (including real values such as the learning rate, or discrete values such as the number of layers).
* [Hyperas](https://github.com/maxpumperla/hyperas), [kopt](https://github.com/Avsecz/kopt) or [Talos](https://github.com/autonomio/talos): optimizing hyperparameters for Keras model (the first two are based on Hyperopt).
* [Scikit-Optimize](https://scikit-optimize.github.io/) (skopt): a general-purpose optimization library. The Bayes SearchCV class performs Bayesian optimization using an interface similar to Grid SearchCV.
* [Spearmint](https://github.com/JasperSnoek/spearmint): a Bayesian optimization library.
* [Sklearn-Deap](https://github.com/rsteca/sklearn-deap): a hyperparameter optimization library based on evolutionary algorithms, also with a GridSearchCV-like interface.
* And many more!

Moreover, many companies offer services for hyperparameter optimization. For example Google Cloud ML Engine has a [hyperparameter tuning service](https://homl.info/googletuning). Other com‐ panies provide APIs for hyperparameter optimization, such as [Arimo](https://arimo.com/), [SigOpt](https://sigopt.com/), [Oscar](http://oscar.calldesk.ai/) and many more.

Hyperparameter tuning is still an active area of research. Evolutionary algorithms are making a comeback lately. For example, check out DeepMind’s excellent [2017 paper](https://homl.info/pbt)[16](#page44), where they jointly optimize a population of models and their hyperparameters. Goo‐ gle also used an evolutionary approach, not just to search for hyperparameters, but also to look for the best neural network architecture for the problem. They call this *AutoML*, and it is already available as a[cloud service](https://cloud.google.com/automl/).Perhaps the days of buildingneural networks manually will soon be over? Check out Google’s [post](https://homl.info/automlpost) on this topic. In fact, evolutionary algorithms have also been used successfully to train individual neu‐ ral networks, replacing the ubiquitous Gradient Descent! See this [2017 post](https://homl.info/neuroevol) by Uber where they introduce their *Deep Neuroevolution* technique.

Despite all this exciting progress, and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter, so you can build a quick prototype, and restrict the search space. Here are a few guidelines for choosing the number of hidden layers and neurons in an MLP, and selecting good values for some of the main hyperparameters.

**

16 “Population Based Training of Neural Networks,” Max Jaderberg et al. (2017).

**

**318** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

**Number of Hidden Layers**

For many problems, you can just begin with a single hidden layer and you will get reasonable results. It has actually been shown that an MLP with just one hidden layer can model even the most complex functions provided it has enough neurons. For a long time, these facts convinced researchers that there was no need to investigate any deeper neural networks. But they overlooked the fact that deep networks have a much higher *parameter e‡ciency* than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data.

To understand why, suppose you are asked to draw a forest using some drawing soft‐ ware, but you are forbidden to use copy/paste. You would have to draw each tree individually, branch per branch, leaf per leaf. If you could instead draw one leaf, copy/paste it to draw a branch, then copy/paste that branch to create a tree, and finally copy/paste this tree to make a forest, you would be finished in no time. Real-world data is often structured in such a hierarchical way and Deep Neural Networks automatically take advantage of this fact: lower hidden layers model low-level struc‐ tures (e.g., line segments of various shapes and orientations), intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces).

Not only does this hierarchical architecture help DNNs converge faster to a good sol‐ ution, it also improves their ability to generalize to new datasets. For example, if you have already trained a model to recognize faces in pictures, and you now want to train a new neural network to recognize hairstyles, then you can kickstart training by reusing the lower layers of the first network. Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the value of the weights and biases of the lower layers of the first network. This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles). This is called *transfer learning*.

In summary, for many problems you can start with just one or two hidden layers and it will work just fine (e.g., you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accu‐ racy using two hidden layers with the same total amount of neurons, in roughly the same amount of training time). For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very com‐ plex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones, as we will see in Chapter 14), and they need a huge amount of training data. However, you will rarely have to train such networks from scratch: it is much more common to



**Fine-Tuning Neural Network Hyperparameters** **|** **319**

reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will be a lot faster and require much less data (we will discuss this in Chap‐ ter 11).

**Number of Neurons per Hidden Layer**

Obviously the number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires 28 x 28 = 784 input neurons and 10 output neurons.

As for the hidden layers, it used to be a common practice to size them to form a pyra‐ mid, with fewer and fewer neurons at each layer—the rationale being that many low-level features can coalesce into far fewer high-level features. For example, a typical neural network for MNIST may have three hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned now, as it seems that simply using the same number of neurons in all hid‐ den layers performs just as well in most cases, or even better, and there is just one hyperparameter to tune instead of one per layer—for example, all hidden layers could simply have 150 neurons. However, depending on the dataset, it can sometimes help to make the first hidden layer bigger than the others.

Just like for the number of layers, you can try increasing the number of neurons grad‐ ually until the network starts overfitting. In general you will get more bang for the buck by increasing the number of layers than the number of neurons per layer. Unfortunately, as you can see, finding the perfect amount of neurons is still somewhat of a dark art.

A simpler approach is to pick a model with more layers and neurons than you actually need, then use early stopping to prevent it from overfitting (and other regu‐ larization techniques, such as *dropout*, as we will see in Chapter 11). This has been dubbed the “stretch pants” approach:[17](#page46) instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size.

**Learning Rate, Batch Size and Other Hyperparameters**

The number of hidden layers and neurons are not the only hyperparameters you can tweak in an MLP. Here are some of the most important ones, and some tips on how to set them:

* The learning rate is arguably the most important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learn‐



17 By Vincent Vanhoucke in his [Deep Learning class](https://homl.info/46) on Udacity.com.



**320** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

ing rate above which the training algorithm diverges, as we saw in Chapter 4 ). So a simple approach for tuning the learning rate is to start with a large value that makes the training algorithm diverge, then divide this value by 3 and try again, and repeat until the training algorithm stops diverging. At that point, you gener‐ ally won’t be too far from the optimal learning rate. That said, it is sometimes useful to reduce the learning rate during training: we will discuss this in Chap‐ ter 11.

* Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important. We will discuss this in Chap‐ ter 11.
* The batch size can also have a significant impact on your model’s performance and the training time. In general the optimal batch size will be lower than 32 (in April 2018, Yann Lecun even tweeted "*Friends don’t let friends use mini-batches* *larger than 32*“). A small batch size ensures that each training iteration is veryfast, and although a large batch size will give a more precise estimate of the gradi‐ ents, in practice this does not matter much since the optimization landscape is quite complex and the direction of the true gradients do not point precisely in the direction of the optimum. However, having a batch size greater than 10 helps take advantage of hardware and software optimizations, in particular for matrix multiplications, so it will speed up training. Moreover, if you use *Batch Normal‐* *ization* (seeChapter 11), the batch size should not be too small (in general no lessthan 20).
* We discussed the choice of the activation function earlier in this chapter: in gen‐ eral, the ReLU activation function will be a good default for all hidden layers. For the output layer, it really depends on your task.
* In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

For more best practices, make sure to read Yoshua Bengio’s great [2012 paper](https://homl.info/138)[18](#page47), which presents many practical recommendations for deep networks.

This concludes this introduction to artificial neural networks and their implementa‐ tion with Keras. In the next few chapters, we will discuss techniques to train very deep nets, we will see how to customize your models using TensorFlow’s lower-level API and how to load and preprocess data efficiently using the Data API, and we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks for sequential data, autoencoders for



18 “Practical recommendations for gradient-based training of deep architectures,” Yoshua Bengio (2012).



**Fine-Tuning Neural Network Hyperparameters** **|** **321**

representation learning, and generative adversarial networks to model and generate data.[19](#page48)

**Exercises**

1. Visit the TensorFlow Playground at [*https://playground.tensorflow.org/*](https://playground.tensorflow.org/)
   * Layers and patterns: try training the default neural network by clicking the run button (top left). Notice how it quickly finds a good solution for the classifica‐ tion task. Notice that the neurons in the first hidden layer have learned simple patterns, while the neurons in the second hidden layer have learned to com‐ bine the simple patterns of the first hidden layer into more complex patterns. In general, the more layers, the more complex the patterns can be.
   * Activation function: try replacing the Tanh activation function with the ReLU activation function, and train the network again. Notice that it finds a solution even faster, but this time the boundaries are linear. This is due to the shape of the ReLU function.
   * Local minima: modify the network architecture to have just one hidden layer with three neurons. Train it multiple times (to reset the network weights, click the reset button next to the play button). Notice that the training time varies a lot, and sometimes it even gets stuck in a local minimum.
   * Too small: now remove one neuron to keep just 2. Notice that the neural net‐ work is now incapable of finding a good solution, even if you try multiple times. The model has too few parameters and it systematically underfits the training set.
   * Large enough: next, set the number of neurons to 8 and train the network sev‐ eral times. Notice that it is now consistently fast and never gets stuck. This highlights an important finding in neural network theory: large neural net‐ works almost never get stuck in local minima, and even when they do these local optima are almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.
   * Deep net and vanishing gradients: now change the dataset to be the spiral (bot‐ tom right dataset under “DATA”). Change the network architecture to have 4 hidden layers with 8 neurons each. Notice that training takes much longer, and often gets stuck on plateaus for long periods of time. Also notice that the neu‐ rons in the highest layers (i.e. on the right) tend to evolve faster than the neu‐ rons in the lowest layers (i.e. on the left). This problem, called the “vanishing gradients” problem, can be alleviated using better weight initialization and



19 A few extra ANN architectures are presented in ???.



**322** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**

other techniques, better optimizers (such as AdaGrad or Adam), or using Batch Normalization.

* + More: go ahead and play with the other parameters to get a feel of what they do. In fact, you should definitely play with this UI for at least one hour, it will grow your intuitions about neural networks significantly.

1. Draw an ANN using the original artificial neurons (like the ones in [Figure 10-3](#page7)) that computes *A* ⊕ *B* (where ⊕ represents the XOR operation). Hint: *A* ⊕ *B* = (*A* ∧ ¬ *B*) ∨ (¬ *A* ∧ *B*).
2. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of threshold logic units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?
3. Why was the logistic activation function a key ingredient in training the first MLPs?
4. Name three popular activation functions. Can you draw them?
5. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activa‐ tion function.
   * What is the shape of the input matrix **X**?
   * What about the shape of the hidden layer’s weight vector **W***h*, and the shape of its bias vector **b***h*?
   * What is the shape of the output layer’s weight vector **W***o*, and its bias vector **b***o*?
   * What is the shape of the network’s output matrix **Y**?
   * Write the equation that computes the network’s output matrix **Y** as a function of **X**, **W***h*, **b***h*, **W***o* and **b***o*.
6. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the out‐ put layer, using what activation function? Answer the same questions for getting your network to predict housing prices as in Chapter 2.
7. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
8. Can you list all the hyperparameters you can tweak in an MLP? If the MLP over‐ fits the training data, how could you tweak these hyperparameters to try to solve the problem?



**Exercises** **|** **323**

1. Train a deep MLP on the MNIST dataset and see if you can get over 98% preci‐ sion. Try adding all the bells and whistles (i.e., save checkpoints, use early stop‐ ping, plot learning curves using TensorBoard, and so on).

Solutions to these exercises are available in ???.



**324** **|** **Chapter 10: Introduction to Arti€cial Neural Networks with Keras**