Chapter 9. Introduction to Artificial Neural Networks

Birds inspired us to fly, burdock plants inspired Velcro, and nature has inspired countless more inventions. It seems only logical, then, to look at the brain's architecture for inspiration on how to build an intelligent machine. This is the logic that sparked *artificial neural networks* (ANNs), machine learning models inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don't have to flap their wings to fly. 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]

ANNs are at the very core of deep learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex machine learning tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple's Siri or Google Assistant) and chatbots (e.g., ChatGPT or Claude), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning how proteins fold (DeepMind's AlphaFold).

This chapter introduces artificial neural networks, starting with a quick tour of the very first ANN architectures and leading up to multilayer perceptrons (MLPs), which are heavily used today (many other architectures will be explored in the following chapters). In this chapter, we will implement simple MLPs using Scikit-Learn to get our feet wet, and in the next chapter we will switch to PyTorch, as it is a much more flexible and efficient library for neural nets.

Now let's go back in time to the origins of artificial neural networks.

From Biological to Artificial 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,[2] "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 you will see.

The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s 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, new architectures were invented and better training techniques were ter. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in *connectionism*, the study of neural networks. But progress was slow, and by the 1990s other powerful machine learning techniques had been invented, such as support vector machines. These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.

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

- 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 possible to train large neural networks in a reasonable amount of time. This is in part due to Moore's law (the number of components in integrated circuits has doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful *graphical processing units* (GPUs) by the millions: GPU cards were initially designed to accelerate graphics, but it turns out that neural networks perform similar computations (such as large matrix multiplications), so they can also be accelerated using GPUs. Moreover, cloud platforms have made this power accessible to everyone.
- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had 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 not a big problem in practice, especially for larger neural networks: the local optima often perform almost as well as the global optimum.
- The invention of the Transformer architecture in 2017 (see Chapter 15) has been a game changer: it can process and generate all sorts of data (e.g., text, images, audio) unlike earlier, more specialized, architectures, and it performs great across a wide variety of tasks from robotics to protein folding. Moreover, it scales rather well, which has made it possible to train very large *foundation models* that can be reused across many different tasks, possibly with a bit of fine-tuning (that's transfer learning), or just by prompting the model in the right way (that's *in-context learning*, or ICL). For instance, you can give it a few examples of the task at hand (that's *few-shot learning*, or FSL), or ask it to reason step-by-step (that's *chain-of-thought* prompting, or CoT). It's a new world!
- 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 progress and even more amazing products. AI is no longer just powering products in the shadows: since chatbots such as ChatGPT were released, the general public is now directly interacting daily with AI assistants, and the big tech companies are competing fiercely to grab this gigantic market: the pace of innovation is wild.

Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron (represented in Figure 9-1). It is an unusual-looking cell mostly found in animal brains. It's composed of a *cell body* containing the nucleus and most of the cell's complex components, 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 *synapses*), which are connected to the dendrites or cell bodies of other neurons.[3] Biological neurons produce short electrical impulses called *action potentials* (APs, or just *signals*), which travel along the axons and make the synapses release chemical signals called *neurotransmitters*. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).

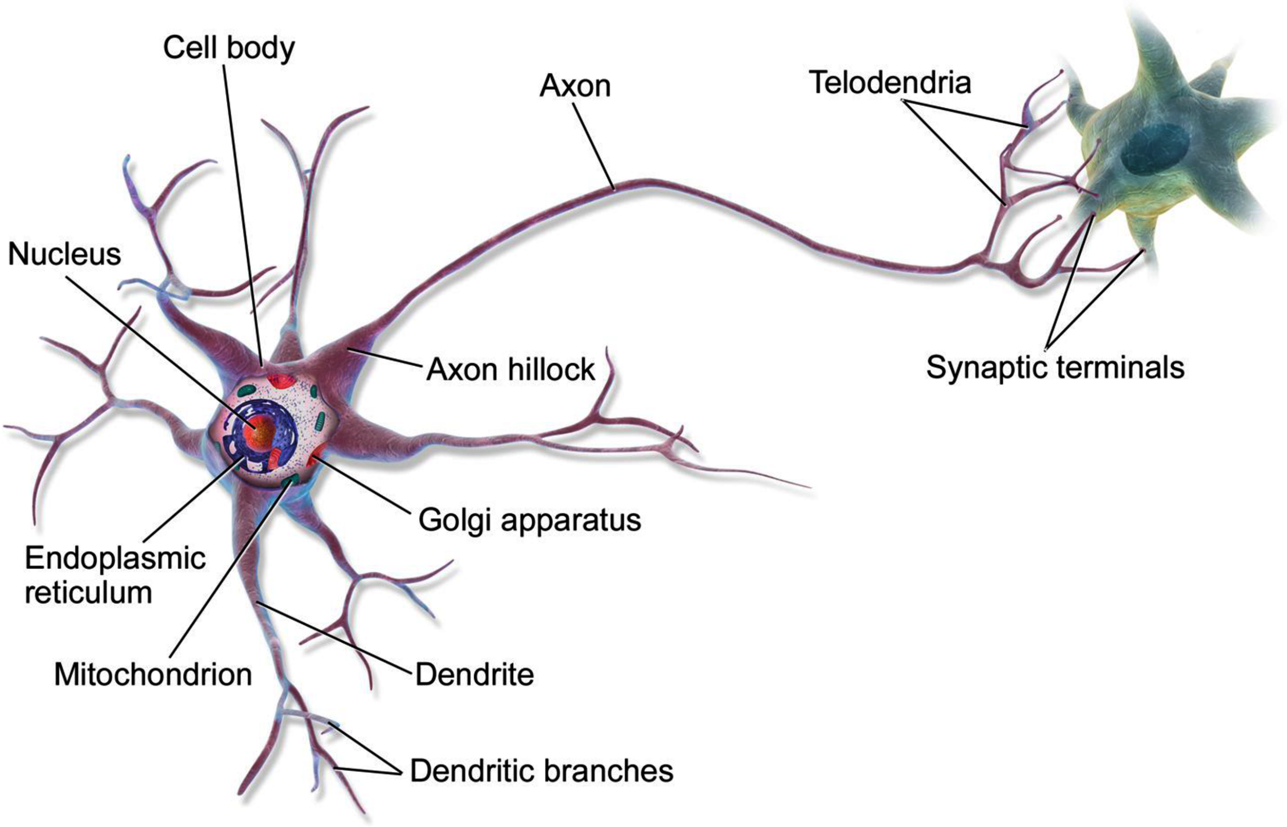


Figure 9-1. A biological neuron[4]

Thus, individual biological neurons seem to behave in a simple way, but they're organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs)[5] is the subject of active research, but some parts of the brain have been mapped. These efforts show that neurons are often organized in consecutive layers, especially in the cerebral cortex (the outer layer of the brain), as shown in Figure 9-2.

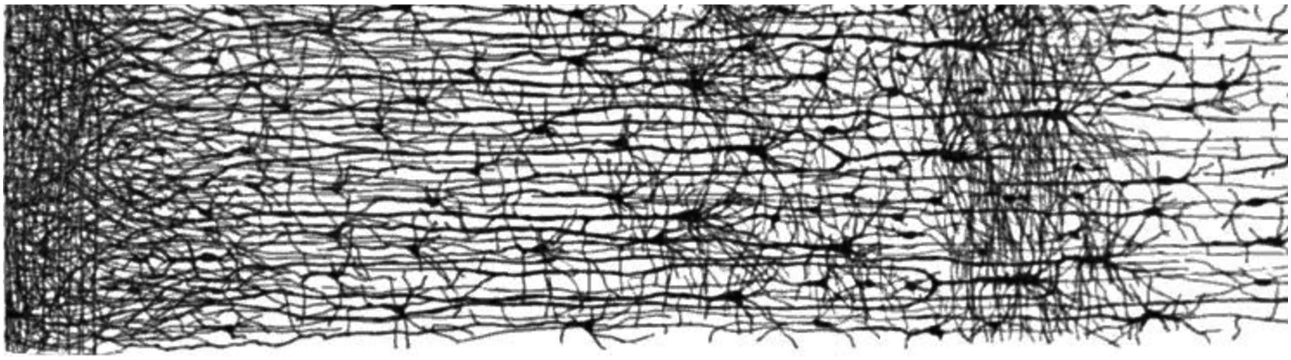


Figure 9-2. Multiple layers in a biological neural network (human cortex)[6]

Logical Computations with Neurons

McCulloch and 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 activates its output when more than a certain number of its inputs are active. In their paper, McCulloch and Pitts showed that even with such a simplified model it is possible to build a network of artificial neurons that can compute any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computations (see Figure 9-3), assuming that a neuron is activated when at least two of its input connections are active.

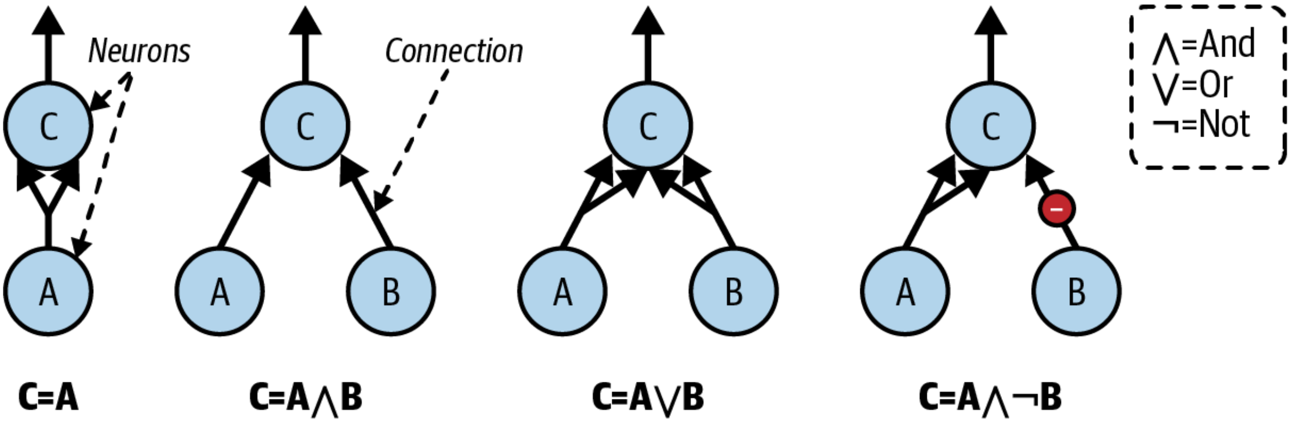


Figure 9-3. ANNs performing simple logical computations

Let's see what these networks do:

- The first network on the left is the identity function: if neuron A is activated, 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 activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron 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 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 imagine how these networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter for an example).

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 9-4) called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU first computes a linear function of its inputs: $z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{w}^\intercal \mathbf{x} + b$. Then it applies a *step function* to the result: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$. So it's almost like logistic regression, except it uses a step function instead of the logistic function.[7] Just like in logistic regression, the model parameters are the input weights $\mathbf{w}$ and the bias term b.

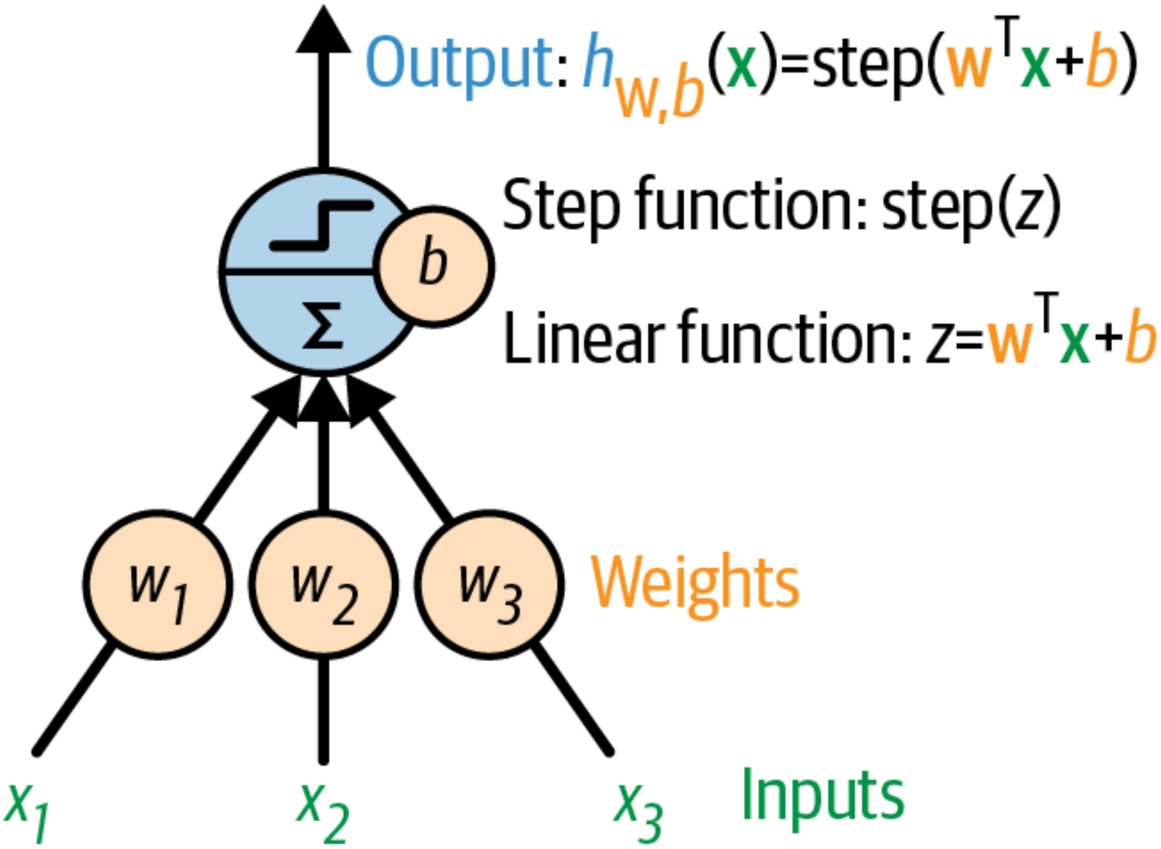

Output: $h_{\mathbf{w},b}(\mathbf{x})$=step($\mathbf{w}^\intercal\mathbf{x}$+$b$)
Step function: step(z)
b
Σ
Linear function: z=$\mathbf{w}^\intercal\mathbf{x}$+$b$
w_1
w_2
w_3
Weights
x_1
x_2
x_3
Inputs

Figure 9-4. TLU: an artificial neuron that computes a weighted sum of its inputs $\mathbf{w}^\intercal \mathbf{x}$, plus a bias term b, then applies a step function

The most common step function used in perceptrons is the *Heaviside step function* (see Equation 9-1). Sometimes the sign function is used instead.

Equation 9-1. Common step functions used in perceptrons (assuming threshold = 0)

A single TLU can be used for simple linear binary classification. It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class. This may remind you of logistic regression (Chapter 4) or linear SVM classification (see the online chapter on SVMs at *https://homl.info/svm*). You could, for example, use a single TLU to classify iris flowers based on petal length and width. Training such a TLU would require finding the right values for w_1, w_2, and b (the training algorithm is discussed shortly).

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a *fully connected layer*, or a *dense layer*. The inputs constitute the *input layer*. And since the layer of TLUs produces the final outputs, it is called the *output layer*. For example, a perceptron with two inputs and three outputs is represented in Figure 9-5.

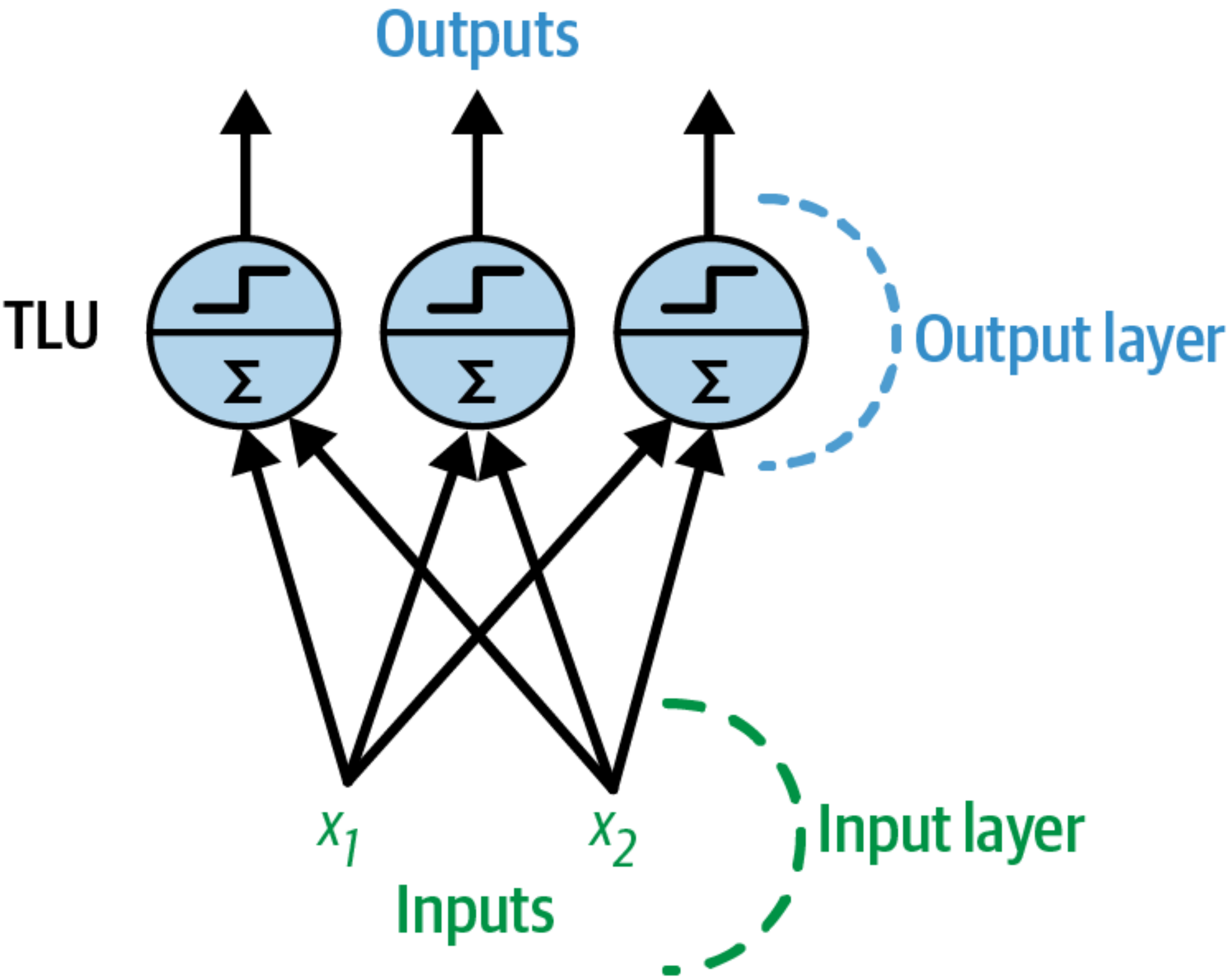


Figure 9-5. Architecture of a perceptron with two inputs and three output neurons

This perceptron can classify instances simultaneously into three different binary classes, which makes it a multilabel classifier. It may also be used for multiclass classification.

Thanks to the magic of linear algebra, Equation 9-2 can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

Equation 9-2. Computing the outputs of a fully connected layer

In this equation:

In this equation:

- $\hat{\mathbf{Y}}$ is the output matrix. It has one row per instance and one column per neuron.
- **X** is the input matrix. It has one row per instance and one column per input feature.
- The weight matrix **W** contains all the connection weights. It has one row per input feature and one column per neuron.[8]
- The bias vector **b** contains all the bias terms: one per neuron.
- The function ϕ is called the *activation function*: when the artificial neurons are TLUs, it is a step function (we will discuss other activation functions shortly).

NOTE

In mathematics, the sum of a matrix and a vector is undefined. However, in data science, we allow "broadcasting": adding a vector to a matrix means adding it to every row in the matrix. So, **XW** + **b** first multiplies **X** by **W**—which results in a matrix with one row per instance and one column per output—then adds the vector **b** to every row of that matrix, which adds each bias term to the corresponding output, for every instance. Moreover, ϕ is then applied itemwise to each item in the resulting matrix.

So, how is a perceptron trained? The perceptron training algorithm proposed by Rosenblatt was largely inspired by *Hebb's rule*. In his 1949 book, *The Organization of Behavior* (Wiley), Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. Siegrid Löwel later summarized Hebb's idea in the catchy phrase, "Cells that fire together, wire together"; that is, the connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule (or *Hebbian learning*). Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the perceptron learning rule 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 9-3.

Equation 9-3. Perceptron learning rule (weight update)

In this equation:

- $w_{i,j}$ is the connection weight between the i^{th} input and the j^{th} neuron.
- x_i is the i^{th} input value of the current training instance.
- is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate (see Chapter 4).

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 will converge to a solution.[9] This is called the *perceptron convergence theorem*.

Scikit-Learn provides a `Perceptron` class that can be used pretty much as you would expect—for example, on the iris dataset (introduced in Chapter 4):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0)  # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new)  # predicts True and False for these 2 flowers
```

You may have noticed that the perceptron learning algorithm strongly resembles stochastic gradient descent (introduced in Chapter 4). 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 regularization).

NOTE

Contrary to logistic regression classifiers, perceptrons do not output a class probability. This is one reason to prefer logistic regression over perceptrons. Moreover, perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as logistic regression or a linear SVM classifier. However, perceptrons may train a bit faster.

In their 1969 monograph, *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 9-6). This is true of any other linear classification model (such as logistic regression classifiers), but researchers had expected much more from perceptrons, and some were so disappointed that they dropped neural networks altogether in favor of more formal approaches such as logic, problem solving, and search. The lack of practical applications also didn't help.

It turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting ANN is called a *multilayer perceptron* (MLP).

The Multilayer Perceptron and Backpropagation

An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the righthand side of Figure 9-6: with inputs (0, 0) or (1, 1), the network outputs 0, and with inputs (0, 1) or (1, 0) it outputs 1. Try verifying that this network indeed solves the XOR problem![10]

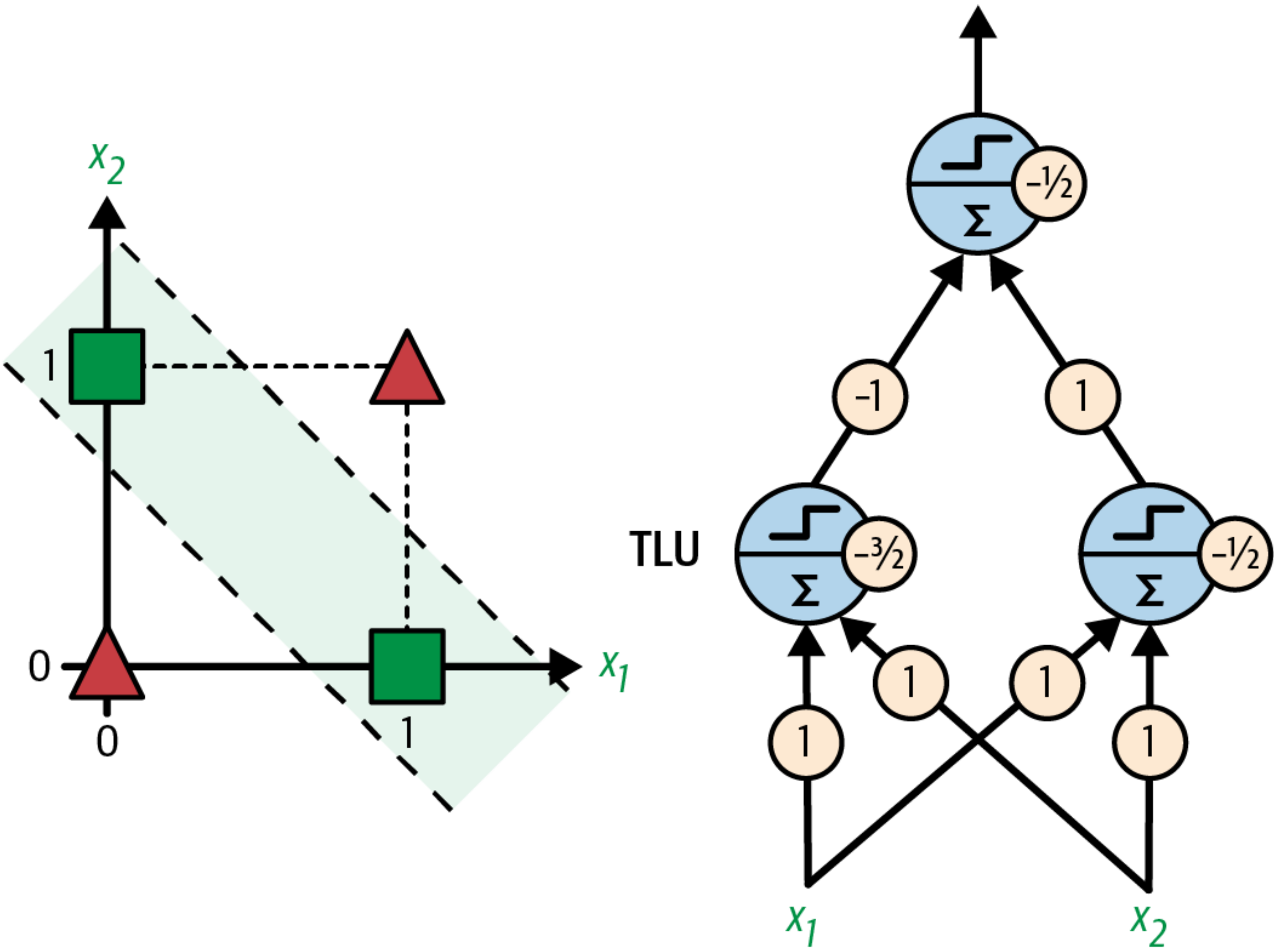
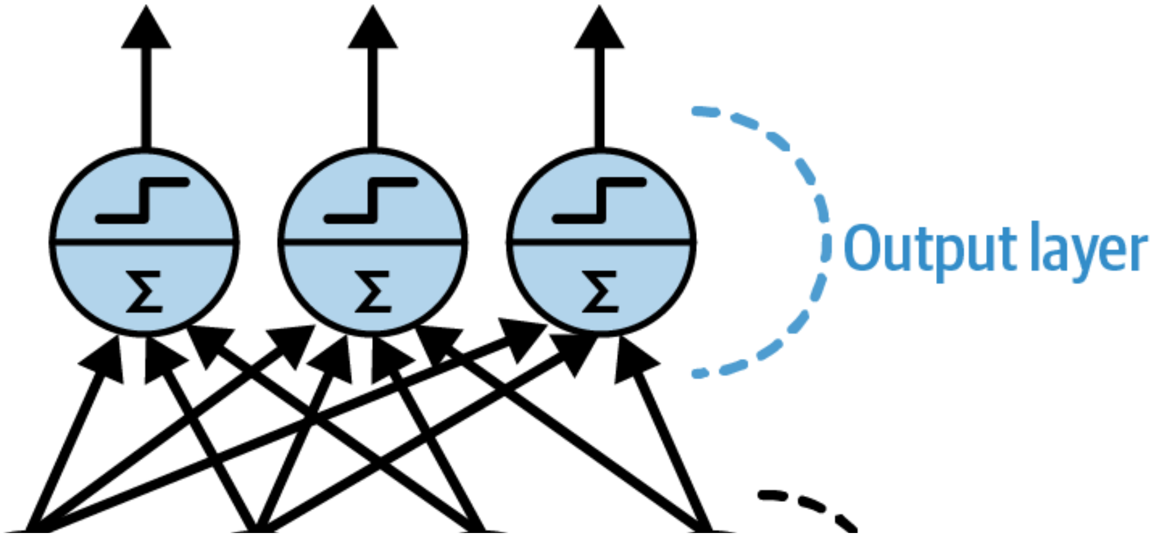


Figure 9-6. XOR classification problem and an MLP that solves it

An MLP is composed of one input layer, one or more layers of artificial neurons (originally TLUs) called *hidden layers*, and one final layer of artificial neurons called the *output layer* (see Figure 9-7). 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*.



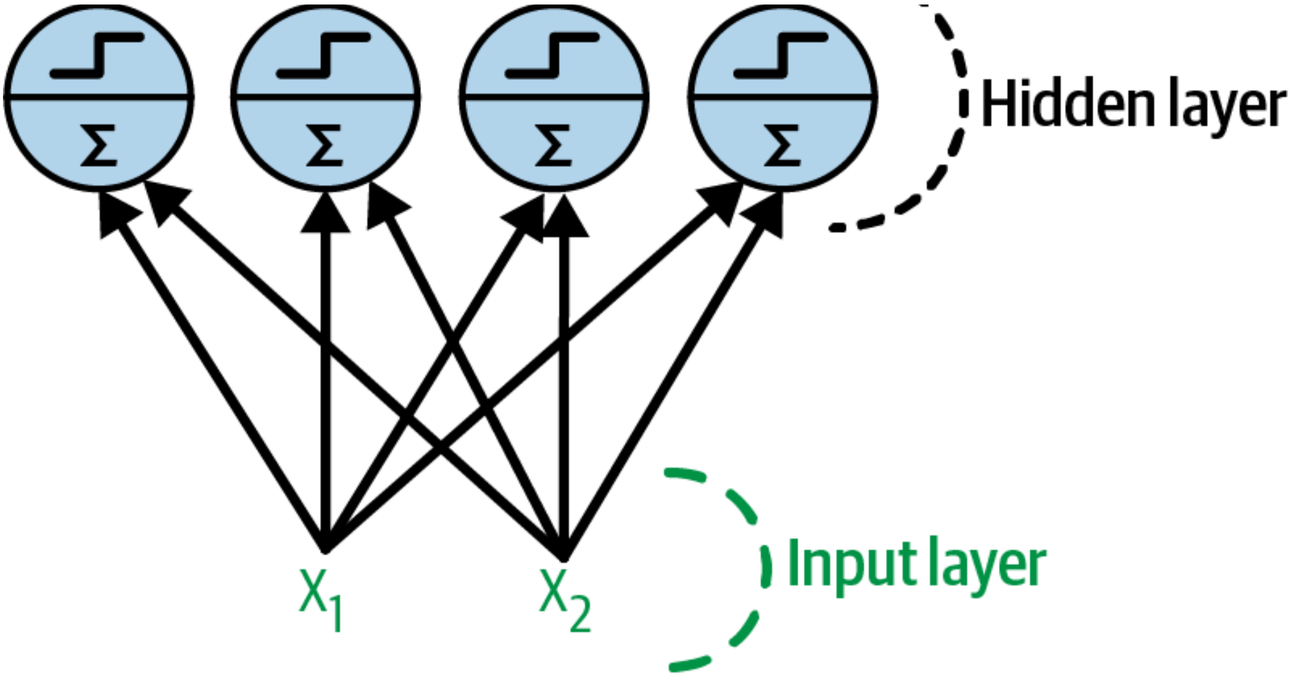


Figure 9-7. Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons

NOTE

The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

When an ANN contains a deep stack of hidden layers,[11] it is called a *deep neural network* (DNN). The field of deep learning studies DNNs, and more generally it is interested in models containing deep stacks of computations. Even so, 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. In the early 1960s several researchers discussed the possibility of using gradient descent to train neural networks, but as we saw in Chapter 4, this requires computing the gradients of the model's error with regard to the model parameters; it wasn't clear at the time how to do this efficiently with such a complex model containing so many parameters, especially with the computers they had back then.

Then, in 1970, a researcher named Seppo Linnainmaa introduced in his master's thesis a technique to compute all the gradients automatically and efficiently. This algorithm is now called *reverse-mode automatic differentiation* (or *reverse-mode autodiff* for short). In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network's error with regard to every single model parameter. In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network's error. These gradients can then be used to perform a gradient descent step. If you repeat this process of computing the gradients automatically and taking a gradient descent step, the neural network's error will gradually drop until it eventually reaches a minimum. This combination of reverse-mode autodiff and gradient descent is now called *backpropagation* (or *backprop* for short).

Here's an analogy: imagine you are learning to shoot a basketball into the hoop. You throw the ball (that's the forward pass), and you observe that it went far off to the right side (that's the error computation), then you consider how you can change your body position to throw the ball a bit less to the right next time (that's the backward pass): you realize that your arm will need to rotate a bit counterclockwise, and probably your whole upper body as well, which in turn means that your feet should turn too (notice how we're going down the "layers"). Once you've thought it through, you actually move your body: that's the gradient descent step. The smaller the errors, the smaller the adjustments. As you repeat the whole process many times, the error gradually gets smaller, and after a few hours of practice, you manage to get the ball through the hoop every time. Good job!

NOTE

There are various autodiff techniques, with different pros and cons. Reverse-mode autodiff is well suited when the function to differentiate has many variables (e.g., connection weights and biases) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out Appendix A.

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: indeed, Linnainmaa's master's thesis was not about neural nets at all, it was more general. It was several more years before backprop started to be used to train neural networks, but it still wasn't mainstream. Then, in 1985, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a paper[12] analyzing how backpropagation allows neural networks to learn useful internal representations. Their results were so impressive that backpropagation was quickly popularized in the field. Over 40 years later, it is still by far the most popular training technique for neural networks.

Let's run through how backpropagation works again in a bit more detail:

- It handles one mini-batch at a time, and goes through the full training set multiple times. If each mini-batch contains 32 instances, and each instance has 100 features, then the mini-batch will be represented as a matrix with 32 rows and 100 columns. Each pass through the training set is called an *epoch*.
- For each mini-batch, the algorithm computes the output of all the neurons in the first hidden layer using Equation 9-2. If the layer has 50 neurons, then its output is a matrix with one row per sample in the mini-batch (e.g., 32), and 50 columns (i.e., one per neuron). This matrix is then 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 function 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 layer parameter contributed to the error. This is done analytically by applying the *chain rule* (one of the most fundamental rules in calcu-

lus), which makes this step fast and precise. The result is one gradient per parameter.

- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights and biases in the 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 connection weights and bias terms in the network, using the error gradients it just computed.

WARNING

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 symmetry* and allow backpropagation to train a diverse team of neurons.

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error (gradient descent step).

In order for backprop to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture: they replaced the step function with the logistic function, $\sigma(z) = 1 / (1 + \exp(-z))$, also called the *sigmoid* function. This was essential because the step function contains only flat segments, so there is no gradient to work with (gradient descent cannot move on a flat surface), while the sigmoid function has a well-defined nonzero derivative everywhere, 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 sigmoid function. Here are two other popular choices:

The hyperbolic tangent function: $\tanh(z) = 2\sigma(2z) - 1$

Just like the sigmoid function, this activation function 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 sigmoid function). That range tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

The rectified linear unit function: $\text{ReLU}(z) = \max(0, z)$

The ReLU function 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 $z < 0$. In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default for most architectures (except the Transformer architecture, as we will see in Chapter 15).[13] Importantly, the fact that it does not have a maximum output value 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 9-8. 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, if f(x) = 2x + 3 and g(x) = 5x – 1, then chaining these two linear functions gives you another linear function: f(g(x)) = 2(5x – 1) + 3 = 10x + 1. So if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

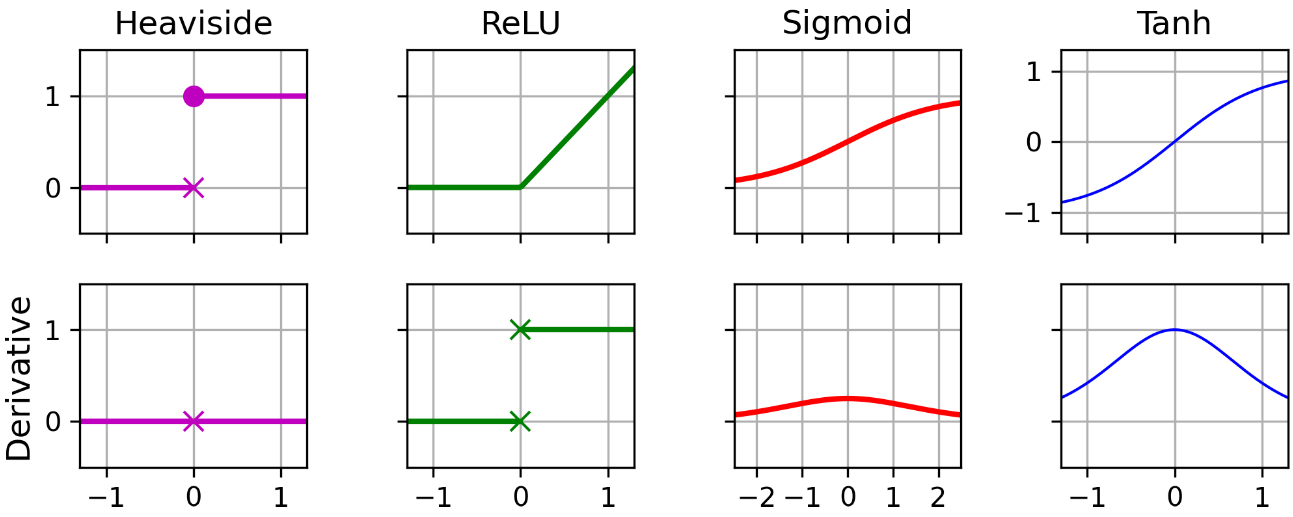


Figure 9-8. Activation functions (left) and their derivatives (right)

OK! You know where neural nets came from, what the MLP architecture looks like, and how it computes its outputs. You've also learned about the backpropagation algorithm. It's time to see MLPs in action!

Building and Training MLPs with Scikit-Learn

MLPs can tackle a wide range of tasks, but the most common are regression and classification. Scikit-Learn can help with both of these. Let's start with regression.

Regression MLPs

How would you build an MLP for a regression task? Well, 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 in an image, you need to predict 2D coordinates, 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 four output neurons.

Scikit-Learn includes an `MLPRegressor` class, so let's use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset. 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 are no missing values. The targets are also scaled down: each unit represents $100,000. Let's start by importing everything we will need:

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import root_mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
```

Next, let's fetch the California housing dataset and split it into a training set and a test set:

```
housing = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
```

Now let's create an `MLPRegressor` model with 3 hidden layers composed of 50 neurons each. The first hidden layer's input size (i.e., the number of rows in its weights matrix) and the output layer's output size (i.e., the number of columns in its weights matrix) will adjust automatically to the dimensionality of the inputs and targets, respectively, when training starts. The model uses the ReLU activation function in all hidden layers, and no activation function at all on the output layer. We also set `verbose=True` to get details on the model's progress during training:

```
mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], early_stopping=True,
                       verbose=True, random_state=42)
```

Since neural nets can have a *lot* of parameters, they have a tendency to overfit the training set. To reduce this risk, it's a good idea to use early stopping (introduced in Chapter 4): when we set `early_stopping=True`, the `MLPRegressor` class automatically sets aside 10% of the training data and uses it to evaluate the model at each epoch (you can adjust the validation set's size by setting `validation_fraction`). If the validation score stops improving for 10 epochs, training automatically stops (you can tweak this number of epochs by setting `n_iter_no_change`).

Now let's create a pipeline to standardize the input features before sending them to the `MLPRegressor`. This is very important because gradient descent does not converge very well when the features have very different scales, as we saw in Chapter 4. We can then train the model! The `MLPRegressor` class uses a variant of gradient descent called *Adam* (see Chapter 11) to minimize the mean squared error. It also uses a tiny bit of ℓ_2 regularization

Chapter 11, to minimize the mean squared error. It also uses a tiny bit of ℓ_2 regularization (you can control its strength via the `alpha` hyperparameter, which defaults to 0.0001):

```
>>> pipeline = make_pipeline(StandardScaler(), mlp_reg)
>>> pipeline.fit(X_train, y_train)
Iteration 1, loss = 0.85190332
Validation score: 0.534299
Iteration 2, loss = 0.28288639
Validation score: 0.651094
[...]
Iteration 45, loss = 0.12960481
Validation score: 0.788517
Validation score did not improve more than tol=0.000100 for 10 consecutive
epochs. Stopping.
```

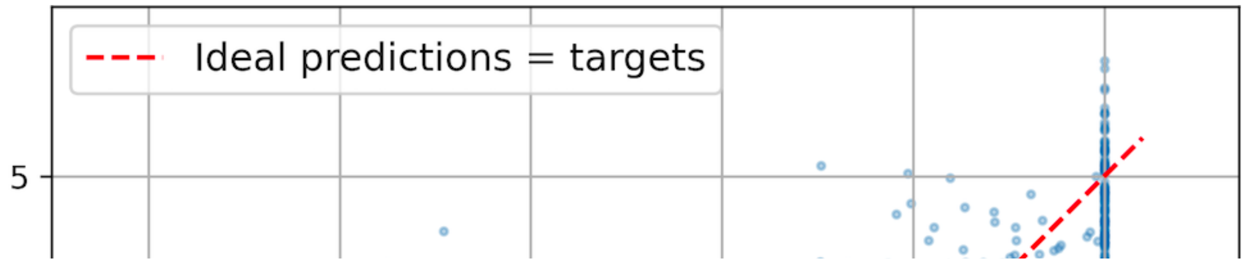
And there you go, you just trained your very first MLP! It required 45 epochs, and as you can see, the training loss went down at each epoch. This loss corresponds to Equation 4-9 divided by 2, so you must multiply it by 2 to get the MSE (although not exactly because the loss includes the ℓ_2 regularization term). The validation score generally went up at each epoch. Like every regressor in Scikit-Learn, `MLPRegressor` uses the R^2 score by default for evaluation—that's what the `score()` method returns. As we saw in Chapter 2, the R^2 score measures the ratio of the variance that is explained by the model. In this case, it reaches close to 80% on the validation set, which is fairly good for this task:

```
>>> mlp_reg.best_validation_score_
0.791536125425778
```

Let's evaluate the RMSE on the test set:

```
>>> y_pred = pipeline.predict(X_test)
>>> rmse = root_mean_squared_error(y_test, y_pred)
>>> rmse
0.5327699946812925
```

We get a test RMSE of about 0.53, which is comparable to what you would get with a random forest classifier. Not too bad for a first try! Figure 9-9 plots the model's predictions versus the targets (on the test set). The dashed red line represents the ideal predictions (i.e., equal to the targets): most of the predictions are close to the targets, but there are still quite a few errors, especially for larger targets.



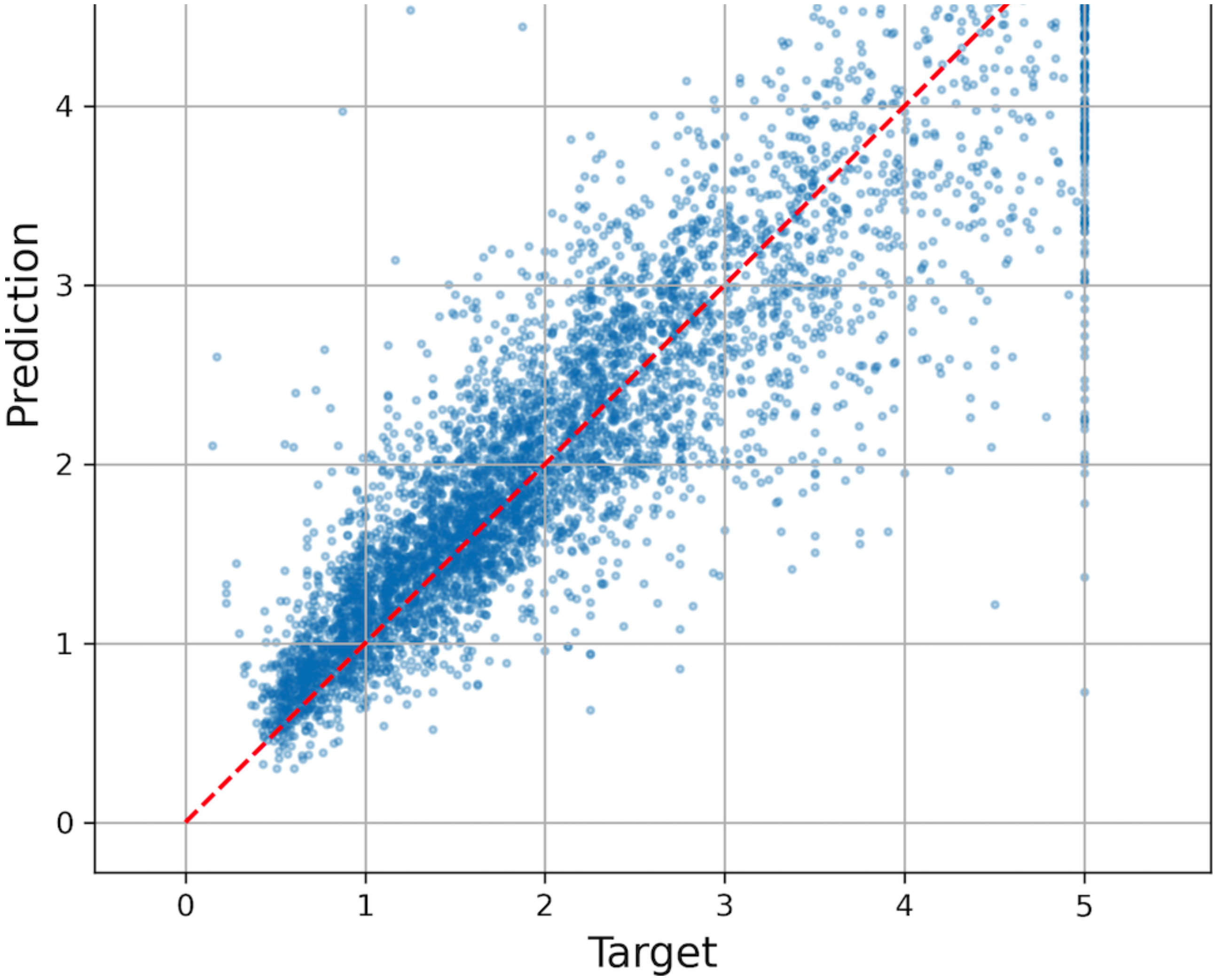


Figure 9-9. MLP regressor’s predictions versus the targets

Note that this MLP does not use any activation function for the output layer, so it’s free to output any value it wants. This is generally fine, but if you want to guarantee that the output is always positive, then you should use the ReLU activation function on the output layer, or the *softplus* activation function, which is a smooth variant of ReLU: softplus(z) = log(1 + exp(z)). Softplus is close to 0 when z is negative, and close to z when z is positive. Finally, if you want to guarantee that the predictions always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and –1 to 1 for tanh. Sadly, the `MLPRegressor` class does not support activation functions in the output layer.

WARNING

Scikit-Learn does not offer GPU acceleration, and its neural net features are fairly limited. This is why we will switch to PyTorch starting in Chapter 10. That said, it is quite convenient to be able to build and train a standard MLP in just a few lines of code using Scikit-Learn: it lets you tackle many complex tasks very quickly.

In general, the mean squared error is the right loss to use for a regression tasks, but if you have a lot of outliers in the training set, you may sometimes prefer to use the mean absolute error instead, or preferably the *Huber loss*, which is a combination of both: it is quadratic when the error is smaller than a threshold δ (typically 1), but linear when the error is larger than δ. The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error. Unfortunately, `MLPRegressor` only supports the MSE loss.

Table 9-1 summarizes the typical architecture of a regression MLP.

Table 9-1. Typical regression MLP architecture

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per target dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

All right, MLPs can tackle regression tasks. What else can they do?

Classification MLPs

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. 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 or nonurgent email. In this case, you would need two output neurons, both using the sigmoid 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 1. This lets the model output any combination of labels: you can have nonurgent ham, urgent ham, nonurgent 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 three 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 softmax activation function for the whole output layer (see Figure 9-10). 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 1, since the classes are exclusive. As we saw in Chapter 3, this is called multiclass classification.

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (or *x-entropy* or log loss for short, see Chapter 4) is generally a good choice.

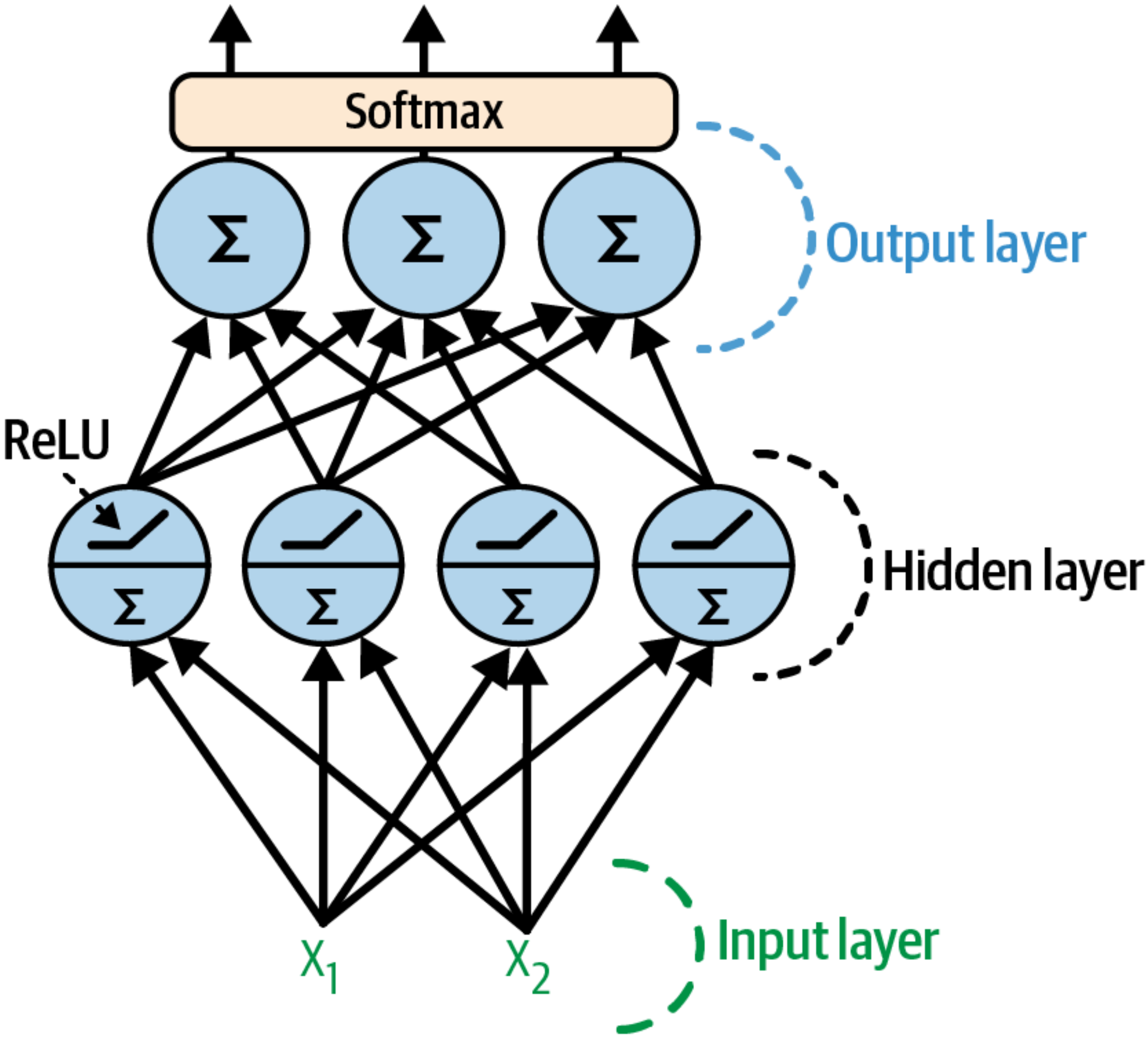


Figure 9-10. A modern MLP (including ReLU and softmax) for classification

Table 9-2 summarizes the typical architecture of a classification MLP.

Table 9-2. Typical classification MLP architecture

Hyperparameter	**Binary classification**	**Multilabel binary classification**	**Multiclass classification**
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy		

As you might expect, Scikit-Learn offers an `MLPClassifier` class in the `sklearn.neural_network` package, which you can use for binary or multiclass classification. It is almost identical to the `MLPRegressor` class, except that its output layer uses the tion. It is almost identical to the `MLPRegressor` class, except that its output layer uses the softmax activation function, and it minimizes the cross-entropy loss rather than the MSE. Moreover, the `score()` method returns the model's accuracy rather than the R^2 score. Let's try it out.

We could tackle the iris dataset, but that task is too simple for a neural net: a linear model would do just as well and wouldn't risk overfitting. So let's instead tackle a more complex task: Fashion MNIST. This 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 much 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. Let's see if we can do better with an MLP.

First, let's load the dataset using the `fetch_openml()` function, very much like we did for MNIST in Chapter 3. Note that the targets are represented as strings `'0'`, `'1'`, …, `'9'`, so we convert them to integers:

```
from sklearn.datasets import fetch_openml

fashion_mnist = fetch_openml(name="Fashion-MNIST", as_frame=False)
targets = fashion_mnist.target.astype(int)
```

The data is already shuffled, so we just take the first 60,000 images for training, and the last 10,000 for testing:

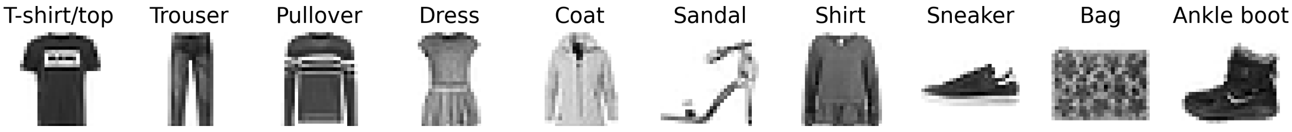
```
X_train, y_train = fashion_mnist.data[:60_000], targets[:60_000]
X_test, y_test = fashion_mnist.data[60_000:], targets[60_000:]
```

Each image is represented as a 1D integer array containing 784 pixel intensities ranging from 0 to 255. You can use the `plt.imshow()` function to plot an image, but first you need to reshape it to `[28, 28]`:

```
import matplotlib.pyplot as plt

X_sample = X_train[0].reshape(28, 28)  # first image in the training set
plt.imshow(X_sample, cmap="binary")
plt.show()
```

If you run this code, you should see the ankle boot represented in the top right corner of Figure 9-11.



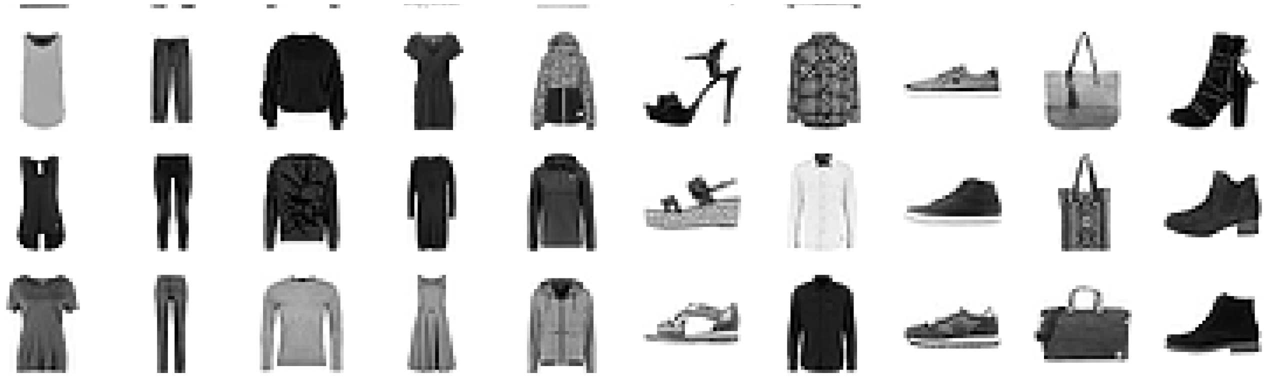


Figure 9-11. First four samples from each class in Fashion MNIST

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. For Fashion MNIST, however, we need the list of class names to know what we are dealing with. Scikit-Learn does not provide it, so let's create it:

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

We can now confirm that the first image in the training set represents an ankle boot:

```
>>> class_names[y_train[0]]
'Ankle boot'
```

We're ready to build the classification MLP:

```
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler

mlp_clf = MLPClassifier(hidden_layer_sizes=[300, 100], verbose=True,
                        early_stopping=True, random_state=42)
pipeline = make_pipeline(MinMaxScaler(), mlp_clf)
pipeline.fit(X_train, y_train)
accuracy = pipeline.score(X_test, y_test)
```

This code is very similar to the regression code we used earlier, but there are a few differences:

- Of course, it's a classification task so we use an `MLPClassifier` rather than an `MLPRegressor`.
- We use just two hidden layers with 300 and 100 neurons, respectively. You can try a different number of hidden layers, and change the number of neurons as well if you want.
- We also use a `MinMaxScaler` instead of a `StandardScaler`. We need it to shrink the pixel intensities down to the 0–1 range rather than 0–255: having features in this range usually works better with the default hyperparameters used by `MLPClassifier`, such as its default learning rate and weight initialization scale. You might wonder why we didn't use a `StandardScaler`? Well, some pixels don't vary much across images; for example, the pix-

StandardScaler? Well some pixels don't vary much across images; for example, the pixels around the edges are almost always white. If we used the StandardScaler, these pixels would get scaled up to have the same variance as every other pixel: as a result, we would give more importance to these pixels than they probably deserve. Using the MinMaxScaler often works better than the StandardScaler for images (but your mileage may vary).
- Lastly, the score() function returns the model's accuracy.

If you run this code, you will find that the model reaches over 90% accuracy on the validation set during training (the exact value is given by mlp_clf.best_validation_score_), but it starts overfitting a bit toward the end, so it ends up at just 89.9% accuracy. When we evaluate the model on the test set, we get 89.3%, which is not bad at all for this task, although we can do better with other neural net architectures such as convolutional neural networks (Chapter 12).

You probably noticed that training was quite slow. That's because the hidden layers have a *lot* of parameters, so there are many computations to run at each iteration. For example, the first hidden layer has 784 × 300 connection weights, plus 300 bias terms, which adds up to 235,500 parameters! All these parameters give the model quite a lot of flexibility to fit the training data, but it also means that there's a high risk of overfitting, especially when you do not have a lot of training data. This is why it's so important to use regularization techniques such as early stopping and ℓ_2 regularization.

Once the model is trained, you can use it to classify new images:

```
>>> X_new = X_test[:15]  # let's pretend these are 15 new images
>>> mlp_clf.predict(X_new)
array([9, 2, 1, 1, 6, 1, 4, 6, 5, 7, 4, 5, 5, 3, 4])
```

All these predictions are correct, except for the one at index 12, which should be a 7 (sneaker) instead of a 5 (sandal). You might want to know how confident the model was about these predictions, especially the bad one. For this, you can use model.predict_proba() instead of model.predict(), like we did in Chapter 3:

```
>>> y_proba = mlp_clf.predict_proba(X_new)
>>> y_proba[12]
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
```

Hmm, that's not great: the model is telling us that it's 100% confident that the image represents a sandal (index 5). So not only is the model wrong, it's 100% confident that it's right. In fact, across all 10,000 images in the test set, there are only 33 images that the model is less than 99.9% confident about, despite the fact that its accuracy is about 90%. That's why you should always treat estimated probabilities with a grain of salt: neural nets have a strong tendency to be overconfident, especially if they are trained for a bit too long.

TIP

The targets for classification tasks can be class indices (e.g., 3) or class probabilities, typically one-hot vectors (e.g., [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]). But if your model tends to be overconfident, you can try the *label smoothing* technique:[14] reduce the target class's probability slightly (e.g., from 1 down to 0.9) and distribute the rest evenly across the other classes (e.g., [0.1/9, 0.1/9, 0.1/9, 0.9, 0.1/9, 0.1/9, 0.1/9, 0.1/9, 0.1/9, 0.1/9]).

Still, getting 90% accuracy on Fashion MNIST is pretty good. You could get even better performance by fine-tuning the hyperparameters, for example using `RandomizedSearchCV`, as we did in Chapter 2. However, the search space is quite large, so it helps to know roughly where to look.

Hyperparameter Tuning Guidelines

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 architecture, but even in a basic MLP you can change the number of layers, the number of neurons and the type of activation function to use in each layer, the weight initialization logic, the type of optimizer to use, its learning rate, the batch size, and more. What are some good values for these hyperparameters?

Number of Hidden Layers

For many problems, you can begin with a single hidden layer and get reasonable results. An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons. But deep networks have a much higher *parameter efficiency* 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. This is because their layered structure enables them to reuse and compose features across multiple levels: for example, the first layer in a face classifier may learn to recognize low-level features such as dots, arcs, or straight lines; while the second layer may learn to combine these low-level features into higher-level features such as squares or circles; and the third layer may learn to combine these higher-level features into a mouth, an eye, or a nose; and the top layer would then be able to use these top-level features to classify faces.

Not only does this hierarchical architecture help DNNs converge faster to a good solution, but 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, you can kickstart the 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 values 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 the neural network will work pretty well. For instance, you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time. For more complex problems, you can ramp up the number of hidden layers until you start overfitting the training set. Very complex 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 you will see in Chapter 12), and they need a huge amount of training data. You will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will then be a lot faster and require much less data.

Number of Neurons per Hidden Layer

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 × 28 = 784 inputs and 10 output neurons.

As for the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer—the rationale being that many low-level features can coalesce into far fewer high-level features. A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned because it seems that using the same number of neurons in all hidden layers performs just as well in most cases, or even better; plus, there is only one hyperparameter to tune, instead of one per layer. That said, depending on the dataset, it can sometimes help to make the first hidden layer a bit larger than the others.

Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. Alternatively, you can try building a model with slightly more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting too much. Vincent Vanhoucke, a Waymo researcher and former Googler, has dubbed this the “stretch pants” approach: 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. With this approach, you avoid bottleneck layers that could ruin your model. Indeed, if a layer has too few neurons, it will lack the computational capacity to model complex relationships, and it may not even have enough representational power to preserve all the useful information from the inputs. For example, if you apply PCA (introduced in Chapter 7) to the Fashion MNIST training set, you will find that you need 187 dimensions to preserve 95% of the variance in the data. So if you set the number of neurons in the first hidden layer to some greater number, say 200, you can be confident that this layer will not be a bottleneck. However, you don’t want to add too many neurons, or else the model will have too many parameters to optimize, and it will take more time and data to train.

TIP

TIP

In general, you will get more bang for your buck by increasing the number of layers rather than the number of neurons per layer.

That said, bottleneck layers are not always a bad thing. For example, limiting the dimensionality of the first hidden layers forces the neural net to keep only the most important dimensions, which can eliminate some of the noise in the data (but don't go too far!). Also, having a bottleneck layer near the output layer can force the neural net to learn good representations of the data in the previous layers (i.e., packing more useful information in less space), which can help the neural net generalize, and can also be useful in and of itself for *representation learning*. We will get back to that in Chapter 18.

Learning Rate

The learning rate is a hugely important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learning rate above which the training algorithm diverges, as we saw in Chapter 4). One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g., $10^{–5}$) and gradually increasing it up to a very large value (e.g., 10). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by $(10 / 10^{-5})^{1 / 500}$ to go from $10^{–5}$ to 10 in 500 iterations). If you plot the loss as a function of the learning rate (using a log scale for the learning rate), you should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up: the optimal learning rate is often a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning point). You can then reinitialize your model and train it normally using this good learning rate.

TIP

To change the learning rate during training when using Scikit-Learn, you must set the MLP's `warm_start` hyperparameter to `True`, and fit the model one batch at a time using `partial_fit()`, much like we did with the `SGDRegressor` in Chapter 4. Simply update the learning rate at each iteration.

Batch Size

The batch size can have a significant impact on your model's performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently (as we will see in Chapter 10), so the training algorithm will see more instances per second. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in *VRAM* (video RAM, i.e., the GPU's memory). There's a catch, though: large batch sizes can sometimes lead to training instabilities, especially with smaller models and at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. Yann LeCun once tweeted "Friends don't let friends use mini-batches larger than 32", citing a 2018 paper[15] by Dominic Masters and Carlo Luschi use mini-batches larger than 32, citing a 2018 paper by Dominic Masters and Carlo Luschi which concluded that using small batches (from 2 to 32) was preferable because small batches led to better models in less training time.

However, other research points in the opposite direction. For example, in 2017, papers by Elad Hoffer et al.[16] and Priya Goyal et al.[17] showed that it is possible to use very large batch sizes (up to 8,192), along with various techniques such as warming up the learning rate (i.e., starting training with a small learning rate, then ramping it up), to obtain very short training times, without any generalization gap.

So one strategy is to use a large batch size, possibly with learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a smaller batch size instead.

Other Hyperparameters

Here are a few more hyperparameters you can tune if you have the computation budget and the time:

Optimizer

> Choosing a better optimizer than plain old mini-batch gradient descent (and tuning its hyperparameters) can help speed up training and sometimes reach better performance.

Activation function

> We discussed how to choose the activation function earlier in this chapter: in general, the ReLU activation function is a good default for all hidden layers. In some cases, replacing ReLU with another function can help.

Number of iterations

> In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

TIP

The optimal learning rate depends on the other hyperparameters—especially the batch size—so if you modify any hyperparameter, make sure to tune the learning rate again.

For more best practices regarding tuning neural network hyperparameters, check out the excellent 2018 paper[18] by Leslie Smith. The Deep Learning Tuning Playbook by Google researchers is also well worth reading. The free e-book *Machine Learning Yearning* by Andrew Ng also contains a wealth of practical advice.

Lastly, I highly recommend you go through exercise 1 at the end of this chapter. You will use a nice web interface to play with various neural network architectures and visualize their outputs. This will be very useful to better understand MLPs and grow a good intuition for the effects of each hyperparameter (number of layers and neurons, activation functions, and more).

This concludes our introduction to artificial neural networks and their implementation with Scikit-Learn. In the next chapter, we will switch to PyTorch, the leading open source library for neural networks, and we will use it to train and run MLPs much faster by exploiting the power of Graphical Processing Units (GPUs). We will also start building more complex models, with multiple inputs and outputs.

Exercises

1. This neural network playground is a great tool to build your intuitions without writing any code (it was built by the TensorFlow team, but there's nothing TensorFlow-specific about it; in fact, it doesn't even use TensorFlow). In this exercise, you will train several binary classifiers in just a few clicks, and tweak the model's architecture and its hyperparameters to gain some intuition on how neural networks work and what their hyperparameters do. Take some time to explore the following:
   a. The patterns learned by a neural net. Try training the default neural network by clicking the Run button (top left). Notice how it quickly finds a good solution for the classification task. The neurons in the first hidden layer have learned simple patterns, while the neurons in the second hidden layer have learned to combine the simple patterns of the first hidden layer into more complex patterns. In general, the more layers there are, the more complex the patterns can be.
   b. Activation functions. Try replacing the tanh activation function with a 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.
   c. The risk of 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.
   d. What happens when neural nets are too small. Remove one neuron to keep just two. Notice that the neural network is now incapable of finding a good solution, even if you try multiple times. The model has too few parameters and systematically underfits the training set.
   e. What happens when neural nets are large enough. Set the number of neurons to eight, and train the network several times. Notice that it is now consistently fast and never gets stuck. This highlights an important finding in neural network theory: large neural networks rarely get stuck in local minima, and even when they do, these local optima are often almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.
   f. The risk of vanishing gradients in deep networks. Select the spiral dataset (the bottom-right dataset under "DATA"), and change the network architecture to have four hidden layers with eight neurons each. Notice that training takes much longer and often gets stuck on plateaus for long periods of time. Also notice that the neurons in the highest

layers (on the right) tend to evolve faster than the neurons in the lowest layers (on the left). This problem, called the *vanishing gradients* problem, can be alleviated with better weight initialization and other techniques, better optimizers (such as AdaGrad or Adam), or batch normalization (discussed in Chapter 11).

   g. Go further. Take an hour or so to play around with other parameters and get a feel for what they do to build an intuitive understanding about neural networks.

2. Draw an ANN using the original artificial neurons (like the ones in Figure 9-3) that computes $A \oplus B$ (where $\oplus$ represents the XOR operation). Hint: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.
3. Why is it generally preferable to use a logistic regression classifier rather than a classic 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?
4. Why was the sigmoid activation function a key ingredient in training the first MLPs?
5. Name three popular activation functions. Can you draw them?
6. 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 activation function.
   a. What is the shape of the input matrix $\mathbf{X}$?
   b. What are the shapes of the hidden layer's weight matrix $\mathbf{W}_h$ and bias vector $\mathbf{b}_h$?
   c. What are the shapes of the output layer's weight matrix $\mathbf{W}_o$ and bias vector $\mathbf{b}_o$?
   d. What is the shape of the network's output matrix $\mathbf{Y}$?
   e. Write the equation that computes the network's output matrix $\mathbf{Y}$ as a function of $\mathbf{X}$, $\mathbf{W}_h$, $\mathbf{b}_h$, $\mathbf{W}_o$, and $\mathbf{b}_o$.
7. 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 output layer, and which activation function should you use? What about for getting your network to predict housing prices, as in Chapter 2?
8. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
9. Can you list all the hyperparameters you can tweak in a basic MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?
10. Train a deep MLP on the CoverType dataset. You can load it using `sklearn.datasets.fetch_covtype()`. See if you can get over 93% accuracy on the test set by fine-tuning the hyperparameters manually and/or using `RandomizedSearchCV`.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

[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.

[2] Warren S. McCulloch and Walter Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity", *The Bulletin of Mathematical Biology* 5, no. 4 (1943): 115–113.

[3] They are not actually attached, just so close that they can very quickly exchange chemical signals.

[4] Image by Bruce Blaus (Creative Commons 3.0). Reproduced from *https://en.wikipedia.org/wiki/Neuron*.

[5] In the context of machine learning, the phrase "neural networks" generally refers to ANNs, not BNNs.

[6] Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from *https://en.wikipedia.org/wiki/Cerebral_cortex*.

[7] Logistic regression and the logistic function were introduced in Chapter 4, along with several other concepts that we will heavily rely on in this chapter, including softmax, cross-entropy, gradient descent, early stopping, and more, so please make sure to read it first.

[8] In some libraries, such as PyTorch, the weight matrix is transposed, so there's one row per neuron, and one column per input feature.

[9] Note that this solution is not unique: when data points are linearly separable, there is an infinity of hyperplanes that can separate them.

[10] For example, when the inputs are (0, 1) the lower-left neuron computes 0 × 1 + 1 × 1 – 3 / 2 = –1 / 2, which is negative, so it outputs 0. The lower-right neuron computes 0 × 1 + 1 × 1 – 1 / 2 = 1 / 2, which is positive, so it outputs 1. The output neuron receives the outputs of the first two neurons as its inputs, so it computes 0 × (–1) + 1 × 1 - 1 / 2 = 1 / 2. This is positive, so it outputs 1.

[11] 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.

[12] David Rumelhart et al., "Learning Internal Representations by Error Propagation" (Defense Technical Information Center technical report, September 1985).

[13] 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 perhaps misleading.

[14] C. Szegedy et al., "Rethinking the Inception Architecture for Computer Vision", CVPR 2016: 2818-2826.

[15] Dominic Masters and Carlo Luschi, "Revisiting Small Batch Training for Deep Neural Networks", arXiv preprint arXiv:1804.07612 (2018).

[16] Elad Hoffer et al., "Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks", *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 1729–1739.

[17] Priya Goyal et al., "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv preprint arXiv:1706.02677 (2017).

[18] Leslie N. Smith, "A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay", arXiv preprint arXiv:1803.09820 (2018).