Chapter 1

Introduction to Artificial Neural Networks

Table of Contents

1.1 Introduction

It is quite apparent that life imitates life and engineers are inspired by nature. 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

Figure 1.1: Nature always is a great source of inspiration for good design. For example, the beak of a bird is aerodynamically efficient and was used in designing the Bullet train [3].The field is of emulating models, systems, and elements of nature for the purpose of solving complex human problems is called biomimetics [25].

(ANN)s, Machine Learning (ML) 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 such as calling them units rather than neurons [1], as some consider this naming to decrease the amount of creativity we can give to the topic .

ANNs are at the very core of **deep learning**. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex ML tasks such as classifying billions of images (e.g., Google Images), powering speech recognition 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 (DeepMind's AlphaGo [13]).

The will treat this chapter as a formal introduction to ANN, starting with a tour of the very first ANN architectures and leading up to multilayer perceptrons, which are heavily used today.

In the second part, we will look at how to implement neural networks using TensorFlow's Keras API. This is a beautifully designed and simple high-level API for building, training, evaluating, and running neural networks. While it may look simple at first glance, it is expressive and flexible enough to let you build a wide variety of neural network architectures.

> For most of your use cases, using `keras` will be enough.

Figure 1.2: The prolific advancements of computers and neural networks have allowed us to tackle problems once deemed impossible. A game of GO requires uncountable amount of moves, yet using ML it was possible to create a software capable of beating the world champion.

1.2 From Biology to Silicon: Artificial Neurons

While it may seem they are the cutting edge in ML, 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 *A Logical Calculus of Ideas Immanent in Nervous Activity*, They 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 [18].

Since then many other architectures have been invented. 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. This is also known as the 1^{st} AI winter [14]. 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 ML techniques had been invented, such as Support Vector Machines (SVM). 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 and this is known as the 2^{nd} AI winter.

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. One of the major turning points of ANN was the fundamental question of:

Is our understanding of the model at fault or is it merely the lack of data to train?

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**, but also thanks to the gaming industry, which has stimulated the production of powerful GPU cards by the millions which have become the norm to train ML instead of CPUs.

Moore's Law

the number of components in integrated circuits has doubled about every 2 years over the last 50 years.

In addition to previous additions, 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 [8], 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.

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.

1.2.1 Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron. It is an unusual-looking cell mostly found in animal brains.

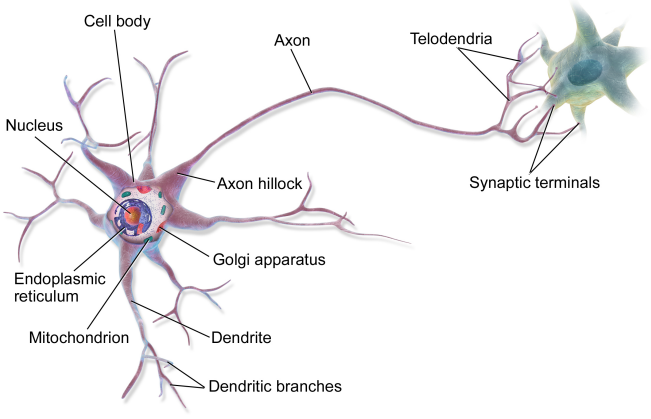

Figure 1.3: A neuron or nerve cell is an excitable cell that fires electric signals called action potentials across a neural network in the nervous system. Neurons communicate with other cells via synapses, which are specialized connections that commonly use minute amounts of chemical neurotransmitters to pass the electric signal from the presynaptic neuron to the target cell through the synaptic gap [22].

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 exten-

sion 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. 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).

Therefore, 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) is the subject of active research, but some parts of the brain have been mapped [2]. These efforts show that neurons are often organized in consecutive layers, especially in the cerebral cortex (the outer layer of the brain).

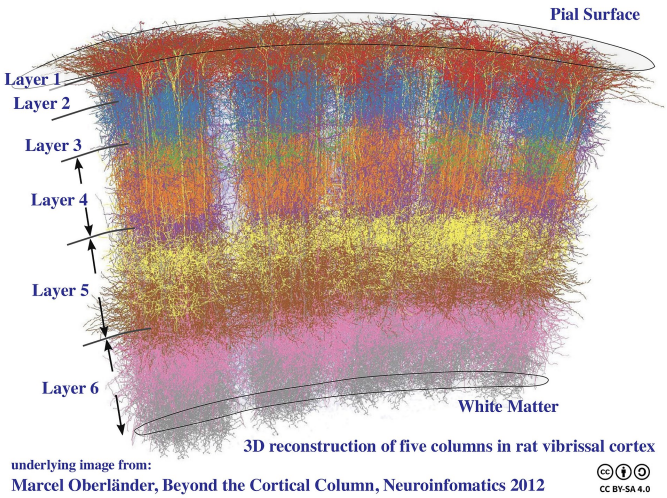

Figure 1.4: A cortical column is a group of neurons forming a cylindrical structure through the cerebral cortex of the brain perpendicular to the cortical surface. The structure was first identified by Vernon Benjamin Mountcastle in 1957. He later identified minicolumns as the basic units of the neocortex which were arranged into columns. Each contains the same types of neurons, connectivity, and firing properties. Columns are also called hypercolumn, macrocolumn, functional column or sometimes cortical module

1.2.2 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 computationsm, assuming that a neuron is activated when at least two of its input connections are active. 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.

1.2.3 The Perceptron

The perceptron is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt [5]. It is based on a slightly different artificial neuron called a Threshold Logic Unit (TLU), or sometimes a Linear Threshold Unit (LTU) which can be seen in Fig. 1.5. The inputs and output are numbers (this is 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:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{x}^T \mathbf{w} + b$$

Then it applies a **step function** to the result:

$$h(x) = \text{step}(z) \qquad \text{where} \qquad z = \mathbf{x}^T \mathbf{w}.$$

It is similar to logistic regression, except it uses a step function instead of the logistic function. Just like in logistic regression, the model parameters are the input weights w and the bias term b.

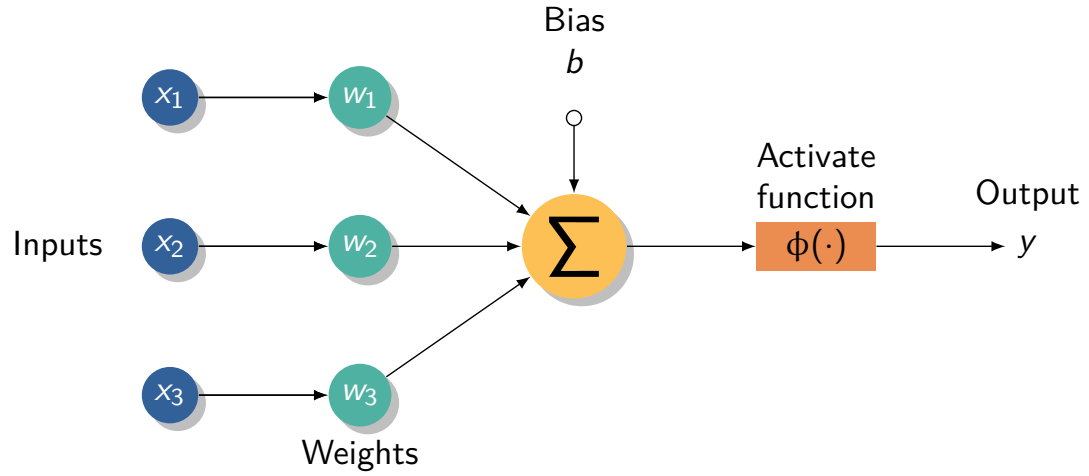

Figure 1.5: Threshold logic unit: an artificial neuron which computes a weighted sum of its inputs then applies a certain activation function.

The most common step function used in perceptrons is the **Heaviside step** and sometimes the **sign function** is used instead [23].

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if} \quad z < 0, \\ 1 & \text{if} \quad z \geq 0. \end{cases} \qquad \text{sgn}(z) = \begin{cases} -1 & \text{if} \quad z < 0, \\ 0 & \text{if} \quad z = 0, \\ +1 & \text{if} \quad z > 0, \end{cases}$$

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 outputs the negative class. They exhibit a similar behaviour to logistic regression or linear SVM classification.*

It is possible, for example, use a single TLU to classify iris flowers [6] (a famous dataset used by statisticians and ML researchers) based on petal length and width. Training such a TLU would require finding the right values for w_1, w_2, b.

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.

This perceptron can classify instances simultaneously into three (3) different binary classes, which makes it a **multilabel classifier**. It may also be used for multiclass classification.

Using linear algebra, the following equation can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

In this equation:

- $\mathbf{X}$ represents the matrix of input features. It has one row per instance and one column per feature.
- $\mathbf{W}$ is the weight matrix containing all the connection weights. It has one row per input and one column per neuron.
- $\mathbf{b}$ is the bias term containing all the bias terms: one per neuron.
- ϕ is the activation function is called the activation function: when the artificial neurons are TLUs, it is a step function.

Now the question is:

> *How is this perceptron train?*

The original perceptron training algorithm proposed by Rosenblatt was largely inspired by Hebb's rule [24]. In his 1949 book *The Organization of Behaviour*, Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger [9].

Siegrid Löwel later summarized Hebb's idea in the catchy phrase,

> *Cells that fire together, wire together*

This means 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 [7])

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.

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta \left(y_j - \hat{y}_j\right) x_i x$$

where:

- $w_{i,j}$ 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.
- $\hat{y}^j$ 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.

The decision boundary of each output neuron is **linear**, therefore perceptrons are incapable of learning complex patterns. However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.

> This is called the perceptron convergence theorem.

```python
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
```

For those of you who have taken a **Data Science II** course, you may have noticed that the perceptron learning algorithm strongly resembles *stochastic gradient descent*. In fact, `sklearn`'s `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters:

- `loss="perceptron"`,
- `learning_rate="constant"`,
- `eta0=1` (the learning rate),
- `penalty=None` (no regularization).

In their 1969 monograph Perceptrons, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of perceptrons: in particular, they are incapable of solving some trivial problems (e.g., the exclusive OR (XOR) classification problem).

XOR Problem

A simple logic gate problem which is proven to be unsolvable using a single-layer perceptron.

This is true of any other linear classification model, but researchers had expected much more from perceptrons, and some were so disappointed, they dropped neural networks altogether in favour of higher-level problems 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 MLP and a MLP can solve the XOR problem [19].

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.

1.2.4 Multilayer Perceptron and Backpropagation

An MLP is composed of one input layer, one or more layers of TLUs called **hidden layers**, and one final layer of TLUs called the output layer. 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.

The signal flows only in one direction (inputs to outputs), so this architecture is an example of a Feedforward Neural Networks (FNN) [4].

When an ANN contains a deep stack of hidden layers, it is called a Deep Neural Networks (DNN). The field of deep learning studies DNNs, and more generally it is interested in models containing deep stacks of computations [21].

For many years researchers struggled to find a way to train MLPs, without success. In the early 1960s several researchers discussed of using gradient descent to train neural networks. This requires computing the gradients of the model's error with regard to the model parameters and at that time, it wasn't clear at the time how to do this

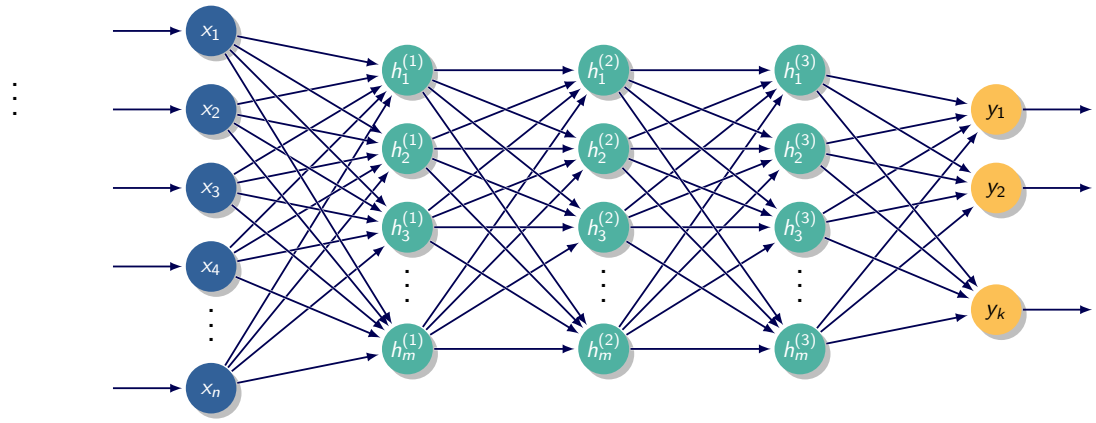

Figure 1.6: Architecture of a Multilayer Perceptron with five inputs, three hidden layer of four neurons, and three output neurons.

efficiently with such a complex model containing so many parameters.

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** [17]. 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. Repeating the 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 automatic differentiation and gradient descent is now called **backpropagation** [10].

> There are various automatic differentiation techniques (i.e., forward and reverse), with each having its own advantages and disadvantages. 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).

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: Linnainmaa's M.Sc thesis was not about neural nets, 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 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper analyzing how backpropagation allowed neural networks to learn useful internal representations [20]. Their results were so impressive that back-propagation was quickly popularized in the field. Today, it is by far the most popular training technique for neural networks.

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

1. It handles one mini-batch at a time, and goes through the full training set multiple times. Each pass is called an **epoch**.
2. Each mini-batch enters the network through the input layer. The algorithm then computes the output of all the neurons in the first hidden 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.
3. Next, the algorithm measures the network's output error. This means, it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error.
4. It then computes how much each output bias and each connection to the output layer contributed to the error. This is done analytically by applying the chain rule, which makes this step fast and precise.
5. 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.
6. Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

> Initialize all the hidden layers' connection weights randomly, or 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 therefore 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 back-propagation 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, which is the gradient descent step.

For back-propagation to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture by replacing the step function with the logistic function:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Which is also called the **sigmoid function**. This was an important improvement as step function contains only flat segments, so there is no gradient to work with, while the sigmoid function has a well-defined nonzero derivative everywhere. In fact, the back-propagation algorithm works well with many other activation functions, not just the sigmoid function.

Here are two (2) other popular choices:

The hyperbolic tangent function **Model**

$$\tanh(z) = 2\sigma(2z) - 1$$

Similar sigmoid function, this activation function is also S-shaped, continuous, and differentiable, but its output value ranges from –1 to 1, instead of 0 to 1 like the sigmoid function.

This bigger 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 **Model**

$$ReLU(z) = max(0, z)$$

It is continuous but unfortunately not differentiable at $z = 0$ as 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.

Importantly, the fact that it does not have a maximum output value helps reduce some issues during gradient descent.

You might wonder what is the point of an activation function, let alone whether it is linear or not? Chaining several linear transformations, gives you only linear transformation. For example:

$$f(x) = 2x + 3 \quad \text{and} \quad g(x) = 5x - 1 \quad \rightarrow \quad f(g(x)) = 2(5x - 1) + 3 = 10x + 1$$

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.

> A large enough DNN with nonlinear activations can theoretically approximate any continuous function.

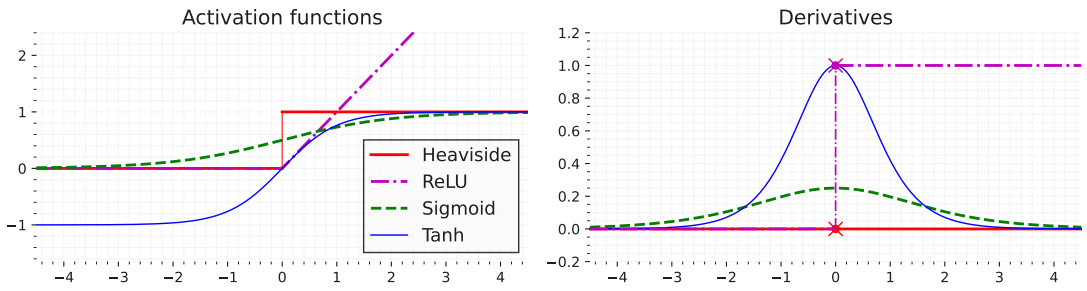

Figure 1.7: The activation function of a node in an ANN is a function which calculates the output of the node based on its individual inputs and their weights. Nontrivial problems can be solved using only a few nodes if the activation function is nonlinear [15]. Modern activation functions include the smooth version of the ReLU, the GELU, which was used in the 2018 BERT model [11], the logistic (sigmoid) function used in the 2012 speech recognition model developed by Hinton et al [12], the ReLU used in the 2012 AlexNet computer vision model [16] and in the 2015 ResNet model.

1.2.5 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), 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. As an example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two (2) 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, in the end you end up with four (4) output neurons.

`sklearn` 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 `sklearn`'s `fetch_california_housing()` function to load the data instead of downloading from a sketchy website.

The following code starts by fetching and splitting the dataset, then it creates a pipeline to standardise the input features before sending them to the `MLPRegressor`. This is very important for neural networks as they are trained using gradient descent, and gradient descent does not converge very well when the features have very different scales.

Finally, the code trains the model and evaluates its validation error. The model uses the ReLU activation function in the hidden layers, and it uses a variant of gradient descent called Adam to minimize the mean squared error, with a little bit of ℓ_2 regularisation:

```python
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import 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

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)
```

We get a validation RMSE of about 0.505, which is comparable to what you would get with a random forest classifier.

> 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 will always be positive, then you should use the ReLU activation function in 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 will 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.

> Building and training a standard MLP with `sklearn` is very convenient, but features are limited. This is why we will switch to Keras in the second part of this chapter.

The `MLPRegressor` class uses the mean squared error, which is usually what you want for regression, but if you have a lot of outliers in the training set, you may prefer to use the mean absolute error instead. Alternatively, you may want to use 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. However, `MLPRegressor` only supports the MSE.

1.2.6 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. 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 10-9). 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 you 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.

`sklearn` has an `MLPClassifier` class in the `sklearn.neural_network` package. It is almost identical to the `MLPRegressor` class, except that it minimizes the cross entropy rather than the MSE. Give it a try now, for example on the iris dataset. It's almost a linear task, so a single layer with 5 to 10 neurons should suffice (make sure to scale the features).

1.3 Implementing MLPs with Keras

Keras is TensorFlow's high-level deep learning API: it allows you to build, train, evaluate, and execute all sorts of neural networks. The original Keras 12 library was developed by François Chollet as part of a research project and was released as a standalone open source project in March 2015. It quickly gained popularity, owing to its ease of use, flexibility, and beautiful design.

Application

Keras used to support multiple backends, including TensorFlow, PlaidML, Theano, and Microsoft Cognitive Toolkit (CNTK) (the last two are sadly deprecated), but since version 2.4, Keras is TensorFlow-only. Similarly, TensorFlow used to include multiple high-level APIs, but Keras was officially chosen as its preferred high-level API when TensorFlow 2 came out. Installing TensorFlow will automatically install Keras as well, and Keras will not work without TensorFlow installed. In short, Keras and TensorFlow fell in love and got married. Other popular deep learning libraries include PyTorch by Facebook and JAX by Google.13

1.3.1 Building an Image Classifier Using Sequential API

Before we do anything, we need to load a dataset. We will use Fashion MNIST. There are 70,000 grayscale images of 28 × 28 pixels each, with 10 classes where images represent fashion items rather than handwritten digits, so each class is more diverse, and the problem turns out to be significantly challenging.

Using Keras to load the dataset

`keras` provides utility functions to fetch and load common datasets, including MNIST, Fashion MNIST, and a few more.

Let's load Fashion MNIST. It's already shuffled and split into a training set (60,000 images) and a test set (10,000 images), but we'll hold out the last 5,000 images from the training set for validation:

```python
import tensorflow as tf

fashion_mnist = tf.keras.datasets.fashion_mnist.load_data()
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist
X_train, y_train = X_train_full[:-5000], y_train_full[:-5000]
X_valid, y_valid = X_train_full[-5000:], y_train_full[-5000:]
```

TensorFlow is usually imported as `tf`, and the Keras API is available via `tf.keras`.

When loading MNIST or Fashion MNIST using `tf.keras` rather than `sklearn`, an important difference is that every image is represented as a 28-by-28 array rather than a 1D array of size 784 with intensities are represented as integers

(from 0 to 255) rather than floats (from 0.0 to 255.0).

Let's take a look at the shape and data type of the training set:

```python
print("The size of the training dataset: ", X_train.shape)
print("The type of the training dataset: ", X_train.dtype)
```

```text
The size of the training dataset:  (55000, 28, 28)
The type of the training dataset:  uint8
```

To make it simple, we'll scale the pixel intensities down to the 0–1 range by dividing them by 255.0

This operation also converts the integer values to floats.

```python
X_train, X_valid, X_test = X_train / 255., X_valid / 255., X_test / 255.
```

Using Fashion MNIST, we need the list of class names to know what we are dealing with:

```python
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 an ankle boot: and below

Figure 1.8: An example of a data within the Fashion MNIST.

we can see some examples of the Fashion MNIST dataset.

1.3.2 Creating the model using the sequential API

It is time to build the neural network. Here is a classification MLP with two (2) hidden layers:

Figure 1.9: A random collection of dataset, making the Fashion MNIST.

```python
tf.random.set_seed(42)
model = tf.keras.Sequential()
model.add(tf.keras.layers.InputLayer(shape=[28, 28]))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(300, activation="relu"))
model.add(tf.keras.layers.Dense(100, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

Let's try to understand the code:

1. Set `tf` random seed to make the results reproducible: the random weights of the hidden layers and the output layer will be the same every time you run your code. You could also choose to use the `tf.keras.utils.set_random_seed()` function, which conveniently sets the random seeds for TensorFlow, Python (`random.seed()`), and NumPy (`np.random.seed()`).

2. Next line creates a *Sequential model*. This is the simplest kind of Keras model for neural networks, composed of a single stack of layers connected sequentially. This is called the sequential API.

3. We build the first layer (an Input layer) and add it to the model. We specify the input shape, which doesn't include the batch size, only the shape of the instances. Keras needs to know the shape of the inputs so it can determine the shape of the connection weight matrix of the first hidden layer.

4. We add a Flatten layer. Its role is to convert each input image into a 1D array: for example, if it receives a batch of shape [32, 28, 28], it will reshape it to [32, 784]. In other words, if it receives input data X, it computes `X.reshape(-1, 784)`. This layer doesn't have any parameters; it's just there to do some simple pre-processing.

5. We add a Dense hidden layer with 300 neurons. It will use the ReLU activation function. Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs. It also manages a vector of bias terms (one per neuron).

6. We add a second Dense hidden layer with 100 neurons, also using the ReLU activation function.

7. We add a Dense output layer with 10 neurons (one per class), using the softmax activation function because the classes are exclusive.

> Writing the argument `activation="relu"` is equivalent to specifying `activation=tf.keras.activations.relu`. Other activation functions are available in the `tf.keras.activations` package.

Instead of adding the layers one by one as we just did, it's often more convenient to pass a list of layers when creating the Sequential model. You can also drop the Input layer and instead specify the `input_shape` in the first layer:

```python
tf.keras.backend.clear_session()
tf.random.set_seed(42)

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=[28, 28]),
    tf.keras.layers.Dense(300, activation="relu"),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(10, activation="softmax")
])
```

The model's `summary()` method displays all the model's layers, including each layer's name, which is automatically generated, its output shape, and its number of parameters.

The summary ends with the total number of parameters, including trainable and non-trainable parameters. Here we only have trainable parameters:

```python
tf.keras.utils.plot_model(model, imagePath+"mnist-_model.pdf", show_shapes=True)
```

Dense layers often have a lot of parameters. For example, the first hidden layer has 784-by-300 connection weights, with 300 bias terms, which adds up to 235,500 parameters.

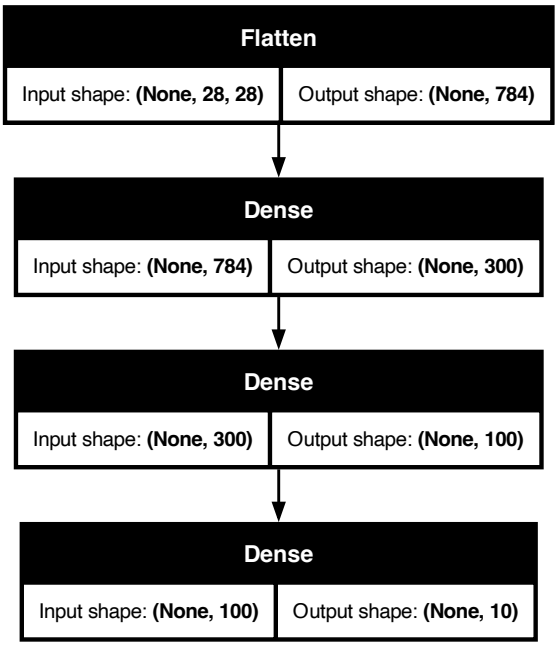

Figure 1.10: The plot of the neural network, showcasing its layers.

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 **over-fitting**, especially when you do not have a lot of training data.

Each layer in a model must have a unique name (e.g., `dense_2`). You can set the layer names explicitly using the constructor's name argument, but generally it's simpler to let Keras name the layers automatically, as we just did. Keras takes the layer's class name and converts it to snake case (i.e., a layer from the `MyCoolLayer` class is named `my_cool_layer` by default). Keras also ensures that the name is **globally unique**, even across models, by appending an index if needed, as in `dense_2`.

This naming scheme makes it possible to merge models easily without getting name conflicts.

> All global state managed by Keras is stored in a Keras session, which you can clear using `tf.keras.backend.clear_session()`.

You can easily get a model's list of layers using the layers attribute, or use the `get_layer()` method to access a layer by name:

```python
print(model.layers)
```

```text
[<Flatten name=flatten, built=True>,
<Dense name=dense, built=True>,
<Dense name=dense_1, built=True>,
<Dense name=dense_2, built=True>]
```

All the parameters of a layer can be accessed using its `get_weights()` and `set_weights()` methods.

For a Dense layer, this includes both the connection weights and the bias terms:

```python
hidden1 = model.layers[1]
weights, biases = hidden1.get_weights()
print(weights)
```

```text
[[-0.05415904  0.00010975 -0.00299759 ...  0.05136904  0.0740822
   0.06472497]
 [ 0.05510217 -0.01353022 -0.00363479 ...  0.07100512 -0.04926914
  -0.02905609]
 [-0.07024231  0.02524897 -0.04784295 ... -0.0521326   0.05084455
  -0.06636713]
 ...
 [ 0.0067075  -0.00256791 -0.064556   ...  0.05266081  0.03520959
  -0.02309504]
 [ 0.05826265 -0.0361187  -0.04228947 ...  0.05612285 -0.03179397
   0.06843598]
 [ 0.06636336 -0.00123435 -0.00247347 ...  0.01809192  0.03434542
   0.00700523]]
```

Notice that the Dense layer initialized the connection weights randomly.

> This is needed to break symmetry.

The biases were initialized to zeros, which is fine.

```python
print(biases)
```

```text
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
```

```text
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

If you want to use a different initialization method, you can set `kernel_initializer` or `bias_initializer` when creating the layer.

Weight Matrix Shape

The shape of the weight matrix depends on the number of inputs, which is why we specified the `input_shape` when creating the model. If you do not specify the input shape, it's OK: Keras will simply wait until it knows the input shape before it actually builds the model parameters. This will happen either when you feed it some data (e.g., during training), or when you call its build() method. Until the model parameters are built, you will not be able to do certain things, such as display the model summary or save the model. So, if you know the input shape when creating the model, it is best to specify it.

Model Compiling

After a model is created, we need to call its `compile()` method to specify the loss function and the optimizer to use, or you can specify a list of extra metrics to compute during training and evaluation:

```python
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="sgd",
              metrics=["accuracy"])
```

Before continuing, we need to explain what is going on here.

We use the `sparse_categorical_crossentropy` 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 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 classification or multilabel binary classification, then we would use the sigmoid activation function in the output layer instead of the softmax activation function, and we would use the `binary_crossentropy` loss.

  Regarding the optimizer, `sgd` means that we will train the model using stochastic gradient descent. Keras will perform the backpropagation algorithm described earlier (i.e., reverse-mode autodiff plus gradient descent).

  Finally, as this is a classifier, it's useful to measure its accuracy during training and evaluation, which is why we set `metrics=["accuracy"]`.

Training and Evaluating Models

Now the model is ready to be trained. For this we simply need to call its `fit()` method:

```
history = model.fit(X_train, y_train, epochs=30,                    python
                    validation_data=(X_valid, y_valid))
```

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 which 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, the model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set.

And that's it! The neural network is trained. At each epoch during training, Keras displays the number of mini-batches processed so far on the left side of the progress bar.

The batch size is 32 by default, and since the training set has 55,000 images, the model goes through 1,719 batches per epoch: 1,718 of size 32, and 1 of size 24.
After the progress bar, you can see the mean training time per sample, and the loss and accuracy (or any other extra metrics you asked for) on both the training set and the validation set and notice that the training loss went down, which is a good sign, and the validation accuracy reached 88.94% after 30 epochs.

That's slightly below the training accuracy, so there is a little bit of overfitting going on, but not a huge amount.
If the training set was very skewed, with some classes being overrepresented and others underrepresented, it would be useful to set the `class_weight` argument when calling the `fit()` method, to give 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, set the `sample_weight` argument. If both `class_weight` and `sample_weight` are provided, then Keras multiplies them. Per-instance weights could be useful, for example, 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 (history.params), the list of epochs it went through (history.epoch), and most importantly 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 any).

```python
print(history.params)
print(history.epoch)
```

```text
{'verbose': 'auto', 'epochs': 30, 'steps': 1719}
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
↪  23, 24, 25, 26, 27, 28, 29]
```

If you use this dictionary to create a Pandas DataFrame and call its `plot()` method, you get the learning curves shown in Fig. 1.11.
You can see that both the training accuracy and the validation accuracy steadily increase during training, while the training loss and the validation loss decrease.

This is good.

The validation curves are relatively close to each other at first, but they get further apart over time, which shows that there's a little bit of overfitting. In this particular case, the model looks like it performed better on the validation set than on the training set at the beginning of training, but that's not actually the case.

The validation error is computed at the end of each epoch, while the training error is

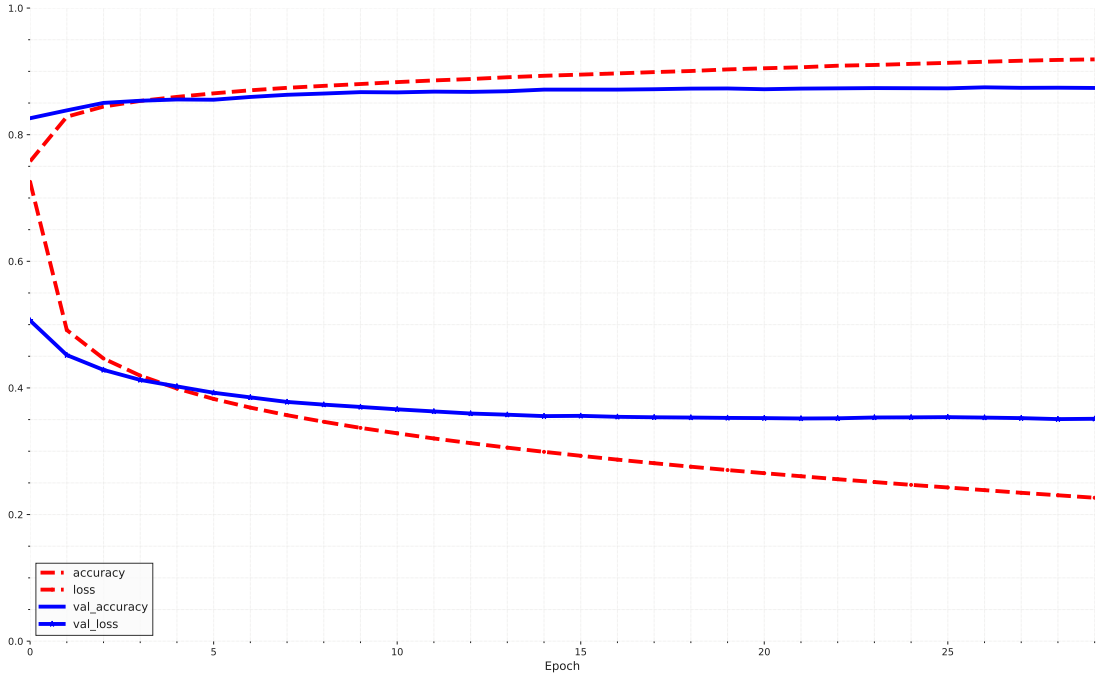

Figure 1.11: Learning curves: the mean training loss and accuracy measured over each epoch, and the mean validation loss and accuracy measured at the end of each epoch

computed using a running mean during each epoch, so the training curve should be shifted by half an epoch to the left.

If you do that, you will see that the training and validation curves overlap almost perfectly at the beginning of training. 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 it would be better to continue training. This is as simple as calling the `fit()` method again, as Keras just continues training where it left off: you should be able to reach about 89.8% validation accuracy, while the training accuracy will continue to rise up to 100%.

> This is not always the case.

If you are not satisfied with the performance of your model, it is a good idea to back and tune the hyperparameters.

1. First check the learning rate (η).
2. If that doesn't help, try another optimizer, and always retune the learning rate after changing any hyperparameter,

3. If the performance is still not great, try tuning model hyperparameters such as the number of layers, the number of neurons per layer, and the types of activation functions to use for each hidden layer.

   You can also try tuning other hyperparameters, such as the batch size (it can be set in the `fit()` method using the `batch_size` argument, which defaults to 32).

   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 `evaluate()` method.

   > It also supports several other arguments, such as `batch_size` and `sample_weight`.

   It is common to get slightly lower performance on the test set than on the validation set, as hyperparameters are **tuned on the validation set**, not the test set. However, in this example, we did not do any hyperparameter tuning, so the lower accuracy is just bad luck.

   > Resist the temptation to tweak the hyperparameters on the test set, or else your estimate of the generalization error will be too optimistic.

Using Model to Make Predictions

It is time to use the model's `predict()` method to make predictions on new instances. As we don't have actual new instances, we'll just use the first three (3) instances of the test set:

```python
X_new = X_test[:3]
y_proba = model.predict(X_new)
print(y_proba.round(2))
```

```text
[[0.   0.   0.   0.   0.   0.12 0.   0.01 0.   0.87]
 [0.   0.   1.   0.   0.   0.   0.   0.   0.   0.  ]
 [0.   1.   0.   0.   0.   0.   0.   0.   0.   0.  ]]
```

For each instance the model estimates one probability per class, from class 0 to class 9. This is similar to the output of the `predict_proba()` method in `sklearn` classifiers.

For example, for the first image it estimates that the probability of class 9 (ankle boot) is 87%, the probability of class 7 (sneaker) is 1%, the probability of class 5 (sandal) is

12%, and the probabilities of the other classes are negligible.

In other words, it is highly confident that the first image is footwear, most likely ankle boots but possibly sneakers or sandals. If you only care about the class with the highest estimated probability (even if that probability is quite low), then you can use the `argmax()` method to get the highest probability class index for each instance:

```python
y_pred = y_proba.argmax(axis=-1)
print(y_pred)
```

```text
[9 2 1]
```

Here, the classifier actually classified all three images correctly, where these images are shown in Fig. 1.12.

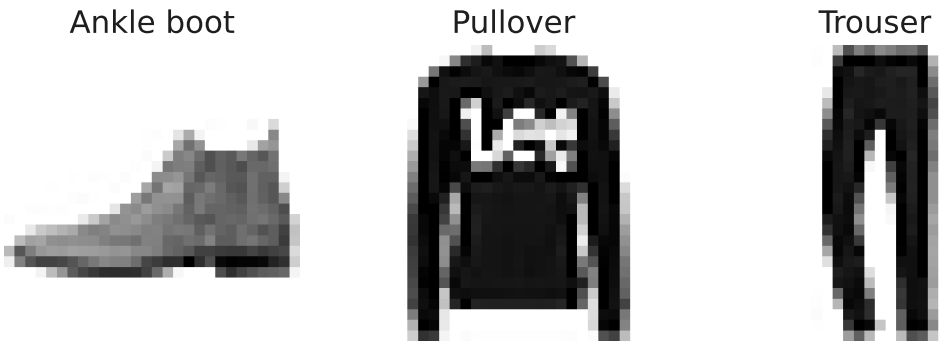

Figure 1.12: Correctly classified Fashion MNIST images.

1.3.3 Building a Regression MLP Using the Sequential API

Instead of classifying categories, lets try to estimate a value. For this application, we need a different dataset. Let's switch back to the California housing problem and tackle it using the same MLP as earlier, with 3 hidden layers composed of 50 neurons each, but this time building it with `tf.keras`.

Using the sequential API to build, train, evaluate, and use a regression MLP is quite similar to what we did for classification. The main differences in the following code example are the fact that the output layer has a single neuron (since we only want to predict a single value) and it uses no activation function, the loss function is the mean squared error, the metric is the RMSE, and we're using an Adam optimizer like `sklearn`s `MLPRegressor` did.

In addition, in this example we don't need a Flatten layer, and instead we're using a Normalization layer as the first layer: it does the same thing as `sklearn`s `StandardScaler`, but it must be fitted to the training data using its `adapt()` method before you call the model's `fit()` method.

Let's look at the code:

```python
housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)
```

```python
tf.random.set_seed(42)
norm_layer = tf.keras.layers.Normalization(input_shape=X_train.shape[1:])
model = tf.keras.Sequential([
    norm_layer,
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(50, activation="relu"),
    tf.keras.layers.Dense(1)
])
optimizer = tf.keras.optimizers.Adam(learning_rate=1e-3)
model.compile(loss="mse", optimizer=optimizer, metrics=["RootMeanSquaredError"])
norm_layer.adapt(X_train)
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
mse_test, rmse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]
y_pred = model.predict(X_new)
```

As you can see, the sequential API is quite clean and straightforward. 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.

Chapter 2

Appendix

Bibliography

[1] S Agatonovic-Kustrin and Rosemary Beresford. “Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research”. In: *Journal of pharmaceutical and biomedical analysis* 22.5 (2000), pp. 717–727.

[2] David GT Barrett, Ari S Morcos, and Jakob H Macke. “Analyzing biological and artificial neural networks: challenges with opportunities for synergy?” In: *Current opinion in neurobiology* 55 (2019), pp. 55–64.

[3] BBC. *How a kingfisher helped reshape Japan’s bullet train*. 2019. URL: https://www.bbc.com/news/av/science-environment-47673287.

[4] George Bebis and Michael Georgiopoulos. “Feed-forward neural networks”. In: *Ieee Potentials* 13.4 (1994), pp. 27–31.

[5] Hans-Dieter Block. “The perceptron: A model for brain functioning. i”. In: *Reviews of Modern Physics* 34.1 (1962), p. 123.

[6] Ronald A Fisher. “The use of multiple measurements in taxonomic problems”. In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[7] Wulfram Gerstner and Werner M Kistler. “Mathematical formulations of Hebbian learning”. In: *Biological cybernetics* 87.5 (2002), pp. 404–415.

[8] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. “Qualitatively characterizing neural network optimization problems”. In: *arXiv preprint arXiv:1412.6544* (2014).

[9] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[10] Robert Hecht-Nielsen. “Theory of the backpropagation neural network”. In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[11] Dan Hendrycks and Kevin Gimpel. “Gaussian error linear units (gelus)”. In: *arXiv preprint arXiv:1606.08415* (2016).

[12] Geoffrey Hinton et al. “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups”. In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[13] Sean D Holcomb et al. “Overview on deepmind and its alphago zero ai”. In: *Proceedings of the 2018 international conference on big data and education*. 2018, pp. 67–71.

[14] Jim Howe. “Artificial intelligence at edinburgh university: A perspective”. In: *Archived from the original on* 17 (2007).

[15] Hinkelmann Knut. "Neural Networks p. 7". In: *University of Applied Sciences Northwestern Switzerland* (2018).

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[17] Seppo Linnainmaa. "Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden Taylor-kehitelmänä". Available in Finnish at `https://people.idsia.ch/~juergen/linnainmaa1970thesis.pdf`. Master's thesis. University of Helsinki, 1970.

[18] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[19] Marius-Constantin Popescu et al. "Multilayer perceptron and neural networks". In: *WSEAS Transactions on Circuits and Systems* 8.7 (2009), pp. 579–588.

[20] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[21] Wojciech Samek et al. "Explaining deep neural networks and beyond: A review of methods and applications". In: *Proceedings of the IEEE* 109.3 (2021), pp. 247–278.

[22] Arnau Sebé-Pedrós. "Stepwise emergence of the neuronal gene expression program in early animal evolution". In: (2023).

[23] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. "Activation functions in neural networks". In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[24] Haim Sompolinsky. "The theory of neural networks: The Hebb rule and beyond". In: *Heidelberg Colloquium on Glassy Dynamics: Proceedings of a Colloquium on Spin Glasses, Optimization and Neural Networks Held at the University of Heidelberg June 9–13, 1986*. Springer. 2006, pp. 485–527.

[25] Julian FV Vincent et al. "Biomimetics: its practice and theory". In: *Journal of the Royal Society Interface* 3.9 (2006), pp. 471–482.

Glossary

ANN Artificial Neural Networks. 1–4, 6, 10, 14

DNN Deep Neural Networks. 10, 14

FNN Feedforward Neural Networks. 10

LTU Linear Threshold Unit. 6

ML Machine Learning. 1–3, 7

MLP Multi-layer Perceptrons. 1, 10, 13–17, 19, 21, 23, 25, 27, 29

SVM Support Vector Machines. 3, 7

TLU Threshold Logic Unit. 6–8, 10