2 The mathematical building blocks of neural networks

This chapter covers

- A first example of a neural network
- Tensors and tensor operations
- How neural networks learn via backpropagation and gradient descent

Understanding deep learning requires familiarity with many simple mathematical concepts: *tensors*, *tensor operations*, *differentiation*, *gradient descent*, and so on. Our goal in this chapter will be to build up your intuition about these notions without getting overly technical. In particular, we'll steer away from mathematical notation, which can introduce unnecessary barriers for those without any mathematics background, and isn't necessary to explain things well. The most precise, unambiguous description of a mathematical operation is its executable code.

To provide sufficient context for introducing tensors and gradient descent, we'll begin the chapter with a practical example of a neural network. Then we'll go over every new concept that's been introduced, point by point. Keep in mind that these concepts will be essential for you to understand the practical examples that will come in the following chapters!

After reading this chapter, you'll have an intuitive understanding of the mathematical theory behind deep learning, and you'll be ready to start diving into modern deep learning frameworks, in chapter 3.

Running the code in this book

This book is full of runnable Python code. Each chapter is paired with a *Jupyter notebook* that contains all of the code from the chapter. A Jupyter notebook is a live Python scratch pad of sorts, where you can interactively run code, graph data, view images, and a lot more. You will gain a lot more practical knowledge from this book if you run and experiment with the code as you read.

By far the easiest way to set up a deep learning environment to run these notebooks is *Google Colaboratory* (or Colab for short), a hosted environment for Jupyter notebooks that has become the industry standard for ML practitioners. With Colab, you can run the code for this book interactively in the browser, connecting to cloud runtimes with configurable hardware. By default, the notebooks in this book will run on Colab's free GPU runtime.

If you would like, you can also run these notebooks locally on your own machine. A GPU is recommended, especially as you get to the larger and more compute-intensive models later in this book.

Instructions for running locally and on Colab, along with the code, can be found at https://github.com/fchollet/deep-learning-with-python-notebooks.

2.1 A first look at a neural network

Let's look at a concrete example of a neural network that uses the machine learning library *Keras* to learn to classify handwritten digits. We will use Keras extensively throughout this book. It's a simple, high-level library that will allow us to stay focused on the concepts we would like to cover.

Unless you already have experience with Keras or similar libraries, you won't understand everything about this first example right away. That's fine. In a few sections, we'll review each element in the example and explain it in detail. So don't worry if some steps seem arbitrary or look like magic to you! We've got to start somewhere.

The problem we're trying to solve here is to classify grayscale images of handwritten digits (28 × 28 pixels) into their 10 categories (0 through 9). We'll use the MNIST dataset, a classic in the machine learning community, which has been around almost as long as the field itself and has been intensively studied. It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s. You can think of "solving" MNIST as the "Hello World" of deep learning—it's what you do to verify that your algorithms are working as expected. As you become a machine learning practitioner, you'll see MNIST come up over and over again, in scientific papers, blog posts, and so on. You can see some MNIST samples in figure 2.1.

NOTE In machine learning, a *category* in a classification problem is called a *class*. Data points are called *samples*. The class associated with a specific sample is called a *label*.

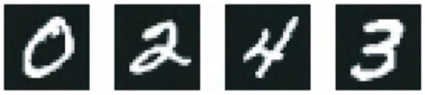
Figure 2.1 MNIST sample digits

The MNIST dataset comes preloaded in Keras, in the form of a set of four NumPy arrays.

Listing 2.1 Loading the MNIST dataset in Keras

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

`train_images` and `train_labels` form the training set, the data that the model will learn from. The model will then be tested on the test set, `test_images` and `test_labels`. The images are encoded as NumPy arrays, and the labels are an array of digits, ranging from 0 to 9. The images and labels have a one-to-one correspondence.

NOTE NumPy is a highly popular Python library for numerical computation. You will see it pop up frequently in your machine learning journey. It is rarely used to implement modern machine learning algorithms, due to lacking GPU and *autodifferentiation* support, but NumPy arrays are often used as a numerical data exchange format—like here, for MNIST digits and their labels.

Let's look at the training data:

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

And here's the test data:

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

The workflow will be as follows. First, we'll feed the neural network the training data, `train_images` and `train_labels`. The network will then learn to associate images and labels. Finally, we'll ask the network to produce predictions for `test_images`, and we'll verify whether these predictions match the labels from `test_labels`.

Let's build the network—again, remember that you aren't expected to understand everything about this example yet.

Listing 2.2 The network architecture

```
import keras
from keras import layers

model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
```

The core building block of neural networks is the *layer*. You can think of a layer as a filter for data: some data goes in, and it comes out in a more useful form. Specifically, layers extract *representations* out of the data fed into them—hopefully, representations that are more meaningful for the problem at hand. Most of deep learning consists of chaining together simple layers that will implement a form of progressive *data distillation*. A deep learning model is like a sieve for data processing, made of a succession of increasingly refined data filters—the layers.

Here, our model consists of a sequence of two `Dense` layers, which are densely connected (also called *fully connected*) neural layers. The second (and last) layer is a 10-way *`softmax` classification* layer, which means it will return an array of 10 probability scores (summing to 1). Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

To make the model ready for training, we need to pick three more things, as part of the *compilation* step:

- *A loss function*—How the model will be able to measure its performance on the training data and thus how it will be able to steer itself in the right direction.
- *An optimizer*—The mechanism through which the model will update itself based on the training data it sees, to improve its performance.
- *Metrics to monitor during training and testing*—Here, we'll only care about accuracy (the fraction of the images that were correctly classified).

The exact purpose of the loss function and the optimizer will be made clear throughout the next two chapters.

Listing 2.3 The compilation step

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

Before training, we'll *preprocess* the data by reshaping it into the shape the model expects and scaling it so that all values are in the `[0, 1]` interval. Previously, our training images were stored in an array of shape `(60000, 28, 28)` of type `uint8` with values in the `[0, 255]` interval. We transform it into a `float32` array of shape `(60000, 28 * 28)` with values between `0` and `1`.

Listing 2.4 Preparing the image data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

We're now ready to train the model, which in Keras is done via a call to the model's `fit()` method—we *fit* the model to its training data.

Listing 2.5 "Fitting" the model

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Two quantities are displayed during training: the loss of the model over the training data and the accuracy of the model over the training data. We quickly reach an accuracy of 0.989 (98.9%) on the training data.

Now that we have a trained model, we can use it to predict class probabilities for *new* digits—images that weren't part of the training data, like those from the test set.

Listing 2.6 Using the model to make predictions

```
>>> test_digits = test_images[0:10]
>>> predictions = model.predict(test_digits)
>>> predictions[0]
array([1.0726176e-10, 1.6918376e-10, 6.1314843e-08, 8.4106023e-06,
       2.9967067e-11, 3.0331331e-09, 8.3651971e-14, 9.9999106e-01,
       2.6657624e-08, 3.8127661e-07], dtype=float32)
```

Each number of index `i` in that array corresponds to the probability that digit image `test_digits[0]` belong to class `i`.

This first test digit has the highest probability score (0.99999106, almost 1) at index 7, so according to our model, it must be a 7:

```
>>> predictions[0].argmax()
7
>>> predictions[0][7]
0.99999106
```

We can check that the test label agrees:

```
>>> test_labels[0]
7
```

On average, how good is our model at classifying such never-before-seen digits? Let's check by computing average accuracy over the entire test set.

Listing 2.7 Evaluating the model on new data

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"test_acc: {test_acc}")
test_acc: 0.9785
```

The test set accuracy turns out to be 97.8%—that's almost double the error rate of the training set (at 98.9% accuracy). This gap between training accuracy and test accuracy is an example of *overfitting*: the fact that machine learning models tend to perform worse on new data than on their training data. Overfitting is a central topic in chapter 5.

This concludes our first example. You just saw how you can build and train a neural network to classify handwritten digits in less than 15 lines of Python code. In this chapter and the next, we'll go into detail about every moving piece we just previewed and clarify what's going on behind the scenes. You'll learn about tensors, the data-storing objects going into the model; tensor operations, which layers are made of; and gradient descent, which allows your model to learn from its training examples.

2.2 Data representations for neural networks

In the previous example, we started from data stored in multidimensional NumPy arrays, also called *tensors*. In general, all current machine learning systems use tensors as their basic data structure. Tensors are fundamental to the field—so fundamental that the TensorFlow framework was named after them. So what's a tensor?

At its core, a tensor is a container for data—usually numerical data. So it's a container for numbers. You may already be familiar with matrices, which are rank-2 tensors: tensors are a generalization of matrices to an arbitrary number of dimensions (note that in the context of tensors, a dimension is often called an *axis*).

Going over the details of tensors might seem a bit abstract at first. But it's well worth it—manipulating tensors will be the bread and butter of any machine learning code you ever write.

2.2.1 Scalars (rank-0 tensors)

A tensor that contains only one number is called a *scalar* (or scalar tensor, rank-0 tensor, or 0D tensor). In NumPy, a `float32` or `float64` number is a scalar tensor (or scalar array). You can display the number of axes of a NumPy tensor via the `ndim` attribute; a scalar tensor has 0 axes (`ndim == 0`). The number of axes of a tensor is also called its *rank*. Here's a NumPy scalar:

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

2.2.2 Vectors (rank-1 tensors)

An array of numbers is called a vector (or rank-1 tensor or 1D tensor). A rank-1 tensor has exactly one axis. The following is a NumPy vector:

```
>>> x = np.array([12, 3, 6, 14, 7])
>>> x
array([12, 3, 6, 14, 7])
>>> x.ndim
1
```

This vector has five entries and so is called a *5-dimensional vector*. Don't confuse a 5D vector with a 5D tensor! A 5D vector has only one axis and has five dimensions along its axis, whereas a 5D tensor has five axes (and may have any number of dimensions along each axis). *Dimensionality* can denote either the number of entries along a specific axis (as in the case of our 5D vector) or the number of axes in a tensor (such as a 5D tensor), which can be confusing at times. In the latter case, it's technically more correct to talk about a *tensor of rank 5* (the rank of a tensor being the number of axes), but the ambiguous notation *5D tensor* is common regardless.

2.2.3 Matrices (rank-2 tensors)

An array of vectors is a *matrix* (or rank-2 tensor or 2D tensor). A matrix has two axes (often referred to as *rows* and *columns*). You can visually interpret a matrix as a rectangular grid of numbers. This is a NumPy matrix:

```
>>> x = np.array([[5, 78, 2, 34, 0],
...               [6, 79, 3, 35, 1],
...               [7, 80, 4, 36, 2]])
```

```
>>> x.ndim
2
```

The entries from the first axis are called the *rows*, and the entries from the second axis are called the *columns*. In the previous example, `[5, 78, 2, 34, 0]` is the first row of `x`, and `[5, 6, 7]` is the first column.

2.2.4 Rank-3 tensors and higher-rank tensors

If you pack such matrices in a new array, you obtain a rank-3 tensor (or 3D tensor), which you can visually interpret as a cube of numbers. The following is a NumPy rank-3 tensor:

```
>>> x = np.array([[[5, 78, 2, 34, 0],
...                 [6, 79, 3, 35, 1],
...                 [7, 80, 4, 36, 2]],
...                [[5, 78, 2, 34, 0],
...                 [6, 79, 3, 35, 1],
...                 [7, 80, 4, 36, 2]],
...                [[5, 78, 2, 34, 0],
...                 [6, 79, 3, 35, 1],
...                 [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

By packing rank-3 tensors in an array, you can create a rank-4 tensor, and so on. In deep learning, you'll generally manipulate tensors with ranks 0 to 4, although you may go up to 5 if you process video data.

2.2.5 Key attributes

A tensor is defined by three key attributes:

- *Number of axes (rank)*—For instance, a rank-3 tensor has three axes, and a matrix has two axes. This is also called the tensor's `ndim` in Python libraries such as NumPy, JAX, TensorFlow, and PyTorch.
- *Shape*—This is a tuple of integers that describes how many dimensions the tensor has along each axis. For instance, the previous matrix example has shape `(3, 5)`, and the rank-3 tensor example has shape `(3, 3, 5)`. A vector has a shape with a single element, such as `(5,)`, whereas a scalar has an empty shape, `()`.
- *Data type (usually called `dtype` in Python libraries)*—This is the type of the data contained in the tensor; for instance, a tensor's type could be `float16`, `float32`, `float64`, `uint8`, `bool`, and so on. In TensorFlow, you are also likely to come across `string` tensors.

To make this more concrete, let's look back at the data we processed in the MNIST example. First, we load the MNIST dataset:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Next, we display the number of axes of the tensor `train_images`, the `ndim` attribute:

```
>>> train_images.ndim
3
```

Here's its shape:

```
>>> train_images.shape
(60000, 28, 28)
```

And this is its data type, the `dtype` attribute:

```
>>> train_images.dtype
uint8
```

So what we have here is a rank-3 tensor of 8-bit integers. More precisely, it's an array of 60,000 matrices of 28 × 28 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255.

Let's display the fourth digit in this rank-3 tensor, using the library Matplotlib (part of the standard scientific Python suite); see figure 2.2.

Listing 2.8 Displaying the fourth digit

```
import matplotlib.pyplot as plt

digit = train_images[4]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

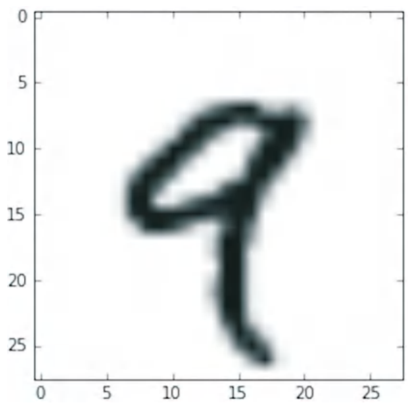

Figure 2.2 The fourth sample in our dataset

Naturally, the corresponding label is just the integer 9:

```
>>> train_labels[4]
9
```

2.2.6 Manipulating tensors in NumPy

In the previous example, we selected a specific digit alongside the first axis using the syntax `train_images[i]`. Selecting specific elements in a tensor is called *tensor slicing*. Let's look at the tensor-slicing operations you can do on NumPy arrays.

The following example selects digits #10 to #100 (#100 isn't included) and puts them in an array of shape `(90, 28, 28)`:

```
>>> my_slice = train_images[10:100]
>>> my_slice.shape
(90, 28, 28)
```

It's equivalent to this more detailed notation, which specifies a start index and stop index for the slice along each tensor axis. Note that `:` is equivalent to selecting the entire axis:

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Equivalent to the previous example

Also equivalent to the previous example

In general, you may select slices between any two indices along each tensor axis. For instance, to select 14 × 14 pixels in the bottom-right corner of all images, you would do this:

```
my_slice = train_images[:, 14:, 14:]
```

It's also possible to use negative indices. Much like negative indices in Python lists, they indicate a position relative to the end of the current axis. To crop the images to patches of 14 × 14 pixels centered in the middle, do this:

```
my_slice = train_images[:, 7:-7, 7:-7]
```

2.2.7 The notion of data batches

In general, the first axis (axis 0, because indexing starts at 0) in all data tensors you'll come across in deep learning will be the *samples axis*. In the MNIST example, "samples" are images of digits.

In addition, deep learning models don't process an entire dataset at once; rather, they break the data into small "batches," or groups of samples with a fixed size. Concretely, here's one batch of our MNIST digits, with a batch size of 128:

```
batch = train_images[:128]
```

And here's the next batch:

```
batch = train_images[128:256]
```

And the nth batch:

```
n = 3
batch = train_images[128 * n : 128 * (n + 1)]
```

When considering such a batch tensor, the first axis (axis 0) is called the *batch axis* (or *batch dimension*). You'll frequently encounter this term when using Keras and other deep learning libraries.

2.2.8 Real-world examples of data tensors

Let's make data tensors more concrete with a few examples similar to what you'll encounter later. The data you'll manipulate will almost always fall into one of the following categories:

- *Vector data*—Rank-2 tensors of shape `(samples, features)`, where each sample is a vector of numerical attributes ("features")
- *Timeseries data or sequence data*—Rank-3 tensors of shape `(samples, timesteps, features)`, where each sample is a sequence (of length `timesteps`) of feature vectors
- *Images*—Rank-4 tensors of shape `(samples, height, width, channels)`, where each sample is a 2D grid of pixels, and each pixel is represented by a vector of values ("channels")
- *Video*—Rank-5 tensors of shape `(samples, frames, height, width, channels)`, where each sample is a sequence (of length `frames`) of images

VECTOR DATA

Vector data is one of the most common cases. In such a dataset, each single data point can be encoded as a vector, and thus a batch of data will be encoded as a rank-2 tensor (that is, an array of vectors), where the first axis is the *samples axis* and the second axis is the *features axis*.

Let's take a look at two examples:

- An actuarial dataset of people, where we consider each person's age, gender, and income. Each person can be characterized as a vector of three values, and thus an entire dataset of 100,000 people can be stored in a rank-2 tensor of shape `(100000, 3)`.

- A dataset of text documents, where we represent each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words). Each document can be encoded as a vector of 20,000 values (one count per word in the dictionary), and thus an entire dataset of 500 documents can be stored in a tensor of shape `(500, 20000)`.

Timeseries data or sequence data

Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a rank-3 tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a rank-2 tensor), and thus a batch of data will be encoded as a rank-3 tensor (see figure 2.3).

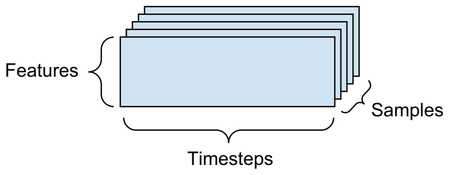

Figure 2.3 A rank-3 timeseries data tensor

The time axis is always the second axis (axis of index 1), by convention. Let's look at a few examples:

- *A dataset of stock prices*—Every minute, we store the current price of the stock, the highest price in the past minute, and the lowest price in the past minute. Thus every minute is encoded as a 3D vector, an entire day of trading is encoded as a matrix of shape `(390, 3)` (there are 390 minutes in a trading day), and 250 days' worth of data can be stored in a rank-3 tensor of shape `(250, 390, 3)`. Here, each sample would be one day's worth of data.
- *A dataset of tweets, where we encode each tweet as a sequence of 280 characters out of an alphabet of 128 unique characters*—In this setting, each character can be encoded as a binary vector of size 128 (an all-zeros vector except for a 1 entry at the index corresponding to the character). Then each tweet can be encoded as a rank-2 tensor of shape `(280, 128)`, and a dataset of 1 million tweets can be stored in a tensor of shape `(1000000, 280, 128)`.

Image data

Images typically have three dimensions: height, width, and color depth. Although grayscale images (like our MNIST digits) have only a single color channel and could thus be stored in rank-2 tensors, by convention image tensors are always rank-3, with a one-dimensional color channel for grayscale images. A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape `(128, 256, 256, 1)`, and a batch of 128 color images could be stored in a tensor of shape `(128, 256, 256, 3)` (see figure 2.4).

There are two conventions for the shapes of image tensors: the *channels-last* convention (which is standard in JAX and TensorFlow, as well as most other deep learning tools out there) and the *channels-first* convention (which is standard in PyTorch).

The channels-last convention places the color-depth axis at the end: `(samples, height, width, color_depth)`. Meanwhile, the channels-first convention places the color depth axis right after the batch axis: `(samples, color_depth, height, width)`.

With the channels-first convention, the previous examples would become `(128, 1, 256, 256)` and `(128, 3, 256, 256)`. The Keras API provides support for both formats.

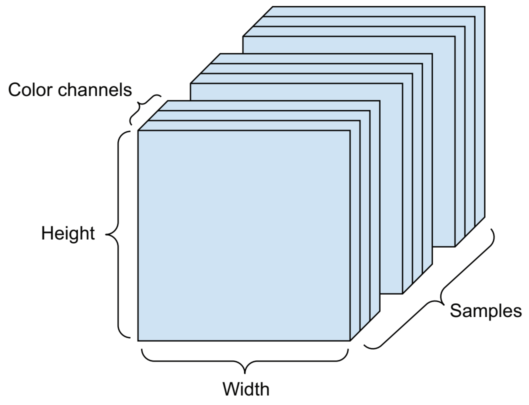

Figure 2.4 A rank-4 image data tensor

VIDEO DATA

Video data is one of the few types of real-world data for which you'll need rank-5 tensors. A video can be understood as a sequence of frames, each frame being a color image. Because each frame can be stored in a rank-3 tensor `(height, width, color_depth)`, a sequence of frames can be stored in a rank-4 tensor `(frames, height, width, color_depth)`, and thus a batch of different videos can be stored in a rank-5 tensor of shape `(samples, frames, height, width, color_depth)`.

For instance, a 60-second, 144 × 256 YouTube video clip sampled at 4 frames per second would have 240 frames. A batch of four such video clips would be stored in a tensor of shape `(4, 240, 144, 256, 3)`. That's a total of 106,168,320 values! If the `dtype` of the tensor was `float32`, then each value would be stored in 32 bits, so the tensor would represent 425 MB. Heavy! Videos you encounter in real life are much lighter because they aren't stored in `float32`, and they're typically compressed by a large factor (such as the MPEG format).

2.3 The gears of neural networks: Tensor operations

Just like any computer program can be ultimately reduced to a small set of binary operations on binary inputs (`AND`, `OR`, `NOR`, and so on), all transformations learned by deep neural networks can be reduced to a handful of *tensor operations* (or *tensor functions*) applied to tensors of numeric data. For instance, it's possible to add tensors, multiply tensors, and so on.

In our initial example, we were building our model by stacking `Dense` layers on top of each other. A Keras layer instance looks like this:

```
keras.layers.Dense(512, activation="relu")
```

This layer can be interpreted as a function, which takes as input a matrix and returns another matrix—a new representation for the input tensor. Specifically, the function is as follows (where `W` is a matrix and `b` is a vector, both attributes of the layer):

```
output = relu(matmul(input, W) + b)
```

Let's unpack this. We have three tensor operations here:

- A tensor product (`matmul`) between the input tensor and a tensor named `W`.
- An addition (`+`) between the resulting matrix and a vector `b`.
- A `relu` operation: `relu(x)` is `max(x, 0)`. `"relu"` stands for "REctified Linear Unit."

NOTE Although this section deals entirely with linear algebra expressions, you won't find any mathematical notation in this book. I've found that mathematical concepts can be more readily mastered by programmers with no mathematical background if they're expressed as short Python snippets instead of mathematical equations. So we'll use NumPy code throughout.

2.3.1 Element-wise operations

The `relu` operation and addition are element-wise operations: operations that are applied independently to each entry in the tensors being considered. This means these operations are highly amenable to massively parallel implementations (*vectorized* implementations, a term that comes from the *vector processor* supercomputer architecture from the 1970–1990 period). If you want to write a naive Python implementation of an element-wise operation, you use a `for` loop, as in this naive implementation of an element-wise `relu` operation:

```python
def naive_relu(x):
    assert len(x.shape) == 2
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

x is a rank-2 NumPy tensor.

Avoids overwriting the input tensor

You could do the same for addition:

```python
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

x and y are rank-2 NumPy tensors.

Avoids overwriting the input tensor

On the same principle, you can do element-wise multiplication, subtraction, and so on.

In practice, when dealing with NumPy arrays, these operations are available as well-optimized built-in NumPy functions, which themselves delegate the heavy lifting to a Basic Linear Algebra Subprograms (BLAS) implementation. BLAS are low-level, highly parallel, efficient tensor-manipulation routines that are typically implemented in Fortran or C.

So, in NumPy, you can do the following element-wise operation, and it will be blazing fast:

```
import numpy as np

z = x + y                  <-- Element-wise addition
z = np.maximum(z, 0.0)     <-- Element-wise relu
```

Let's actually time the difference:

```
import time

x = np.random.random((20, 100))
y = np.random.random((20, 100))

t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.0)
print("Took: {0:.2f} s".format(time.time() - t0))
```

This takes 0.02 seconds. Meanwhile, the naive version takes a stunning 2.45 seconds:

```
t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)
    z = naive_relu(z)
print("Took: {0:.2f} s".format(time.time() - t0))
```

Likewise, when running JAX/TensorFlow/PyTorch code on a GPU, element-wise operations are executed via fully vectorized CUDA implementations that can best utilize the highly parallel GPU chip architecture.

2.3.2 *Broadcasting*

Our earlier naive implementation of `naive_add` only supports the addition of rank-2 tensors with identical shapes. But in the `Dense` layer introduced earlier, we added a rank-2 tensor with a vector. What happens with addition when the shapes of the two tensors being added differ?

When possible, and if there's no ambiguity, the smaller tensor will be *broadcast* to match the shape of the larger tensor. Broadcasting consists of two steps:

- Axes (called *broadcast axes*) are added to the smaller tensor to match the `ndim` of the larger tensor.

- The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.

Let's look at a concrete example. Consider `X` with shape `(32, 10)` and `y` with shape `(10,)`:

```
import numpy as np

X = np.random.random((32, 10))
y = np.random.random((10,))
```

X is a random matrix with shape (32, 10).

y is a random vector with shape (10,).

First, we add an empty first axis to `y`, whose shape becomes `(1, 10)`:

```
y = np.expand_dims(y, axis=0)
```

The shape of y is now (1, 10).

Then, we repeat `y` 32 times alongside this new axis, so that we end up with a tensor `Y` with shape `(32, 10)`, where `Y[i, :] == y` for `i` in `range(0, 32)`:

```
Y = np.tile(y, (32, 1))
```

Repeat y 32 times along axis 0 to obtain Y with shape (32, 10).

At this point, we can add `X` and `Y` because they have the same shape.

In terms of implementation, no new rank-2 tensor is created because that would be terribly inefficient. The repetition operation is entirely virtual: it happens at the algorithmic level rather than at the memory level. But thinking of the vector being repeated 32 times alongside a new axis is a helpful mental model. Here's what a naive implementation would look like:

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

x is a rank-2 NumPy tensor.

y is a NumPy vector.

Avoids overwriting the input tensor

With broadcasting, you can generally apply two-tensor element-wise operations if one tensor has shape `(a, b, … n, n + 1, … m)` and the other has shape `(n, n + 1, … m)`. The broadcasting will then automatically happen for axes `a` through `n - 1`.

The following example applies the element-wise `maximum` operation to two tensors of different shapes via broadcasting:

```
import numpy as np

x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)
```

x is a random tensor with shape (64, 3, 32, 10).

y is a random tensor with shape (32, 10).

The output z has shape (64, 3, 32, 10) like x.

2.3.3 Tensor product

The *tensor product*, also called *dot product* or *matmul* (short for "matrix multiplication") is one of the most common, most useful tensor operations.

In NumPy, a tensor product is done using the `np.matmul` function, and in Keras, with the `keras.ops.matmul` function. Its shorthand is the `@` operator in Python:

```
x = np.random.random((32,))
y = np.random.random((32,))

z = np.matmul(x, y)
z = x @ y
```

Takes the product between x and y

This is equivalent.

In mathematical notation, you'd note the operation with a dot (•) (hence the name "dot product"):

```
z = x • y
```

Mathematically, what does the `matmul` operation do? Let's start with the product of two vectors `x` and `y`. It's computed as follows:

```
def naive_vector_product(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.0
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

x and y are NumPy vectors.

You'll have noticed that the product between two vectors is a scalar and that only vectors with the same number of elements are compatible for this operation.

You can also take the product between a matrix x and a vector y, which returns a vector where the coefficients are the products between y and the rows of x. You implement it as follows:

```
def naive_matrix_vector_product(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

x is a NumPy matrix.

y is a NumPy vector.

The 1st dimension of x must equal the 0th dimension of y!

This operation returns a vector of 0s with as many rows as x.

You could also reuse the code we wrote previously, which highlights the relationship between a matrix-vector product and a vector product:

```
def naive_matrix_vector_product(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_product(x[i, :], y)
    return z
```

Note that as soon as one of the two tensors has an `ndim` greater than 1, `matmul` is no longer *symmetric*, which is to say that `matmul(x, y)` isn't the same as `matmul(y, x)`.

Of course, a tensor product generalizes to tensors with an arbitrary number of axes. The most common applications may be the product between two matrices. You can take the product of two matrices x and y (`matmul(x, y)`) if and only if `x.shape[1] == y.shape[0]`. The result is a matrix with shape `(x.shape[0], y.shape[1])`, where the coefficients are the vector products between the rows of x and the columns of y. Here's the naive implementation:

```
def naive_matrix_product(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
```

x and y are NumPy matrices.

The 1st dimension of x must equal the 0th dimension of y!

This operation returns a matrix of 0s with a specific shape.

Iterates over the rows of x …

… and over the columns of y.

```
            z[i, j] = naive_vector_product(row_x, column_y)
    return z
```

To understand vector product shape compatibility, it helps to visualize the input and output tensors by aligning them as shown in figure 2.5.

x, y, and z are pictured as rectangles (literal boxes of coefficients). Because the rows of x and the columns of y must have the same size, it follows that the width of x must match the height of y. If you go on to develop new machine learning algorithms, you'll likely be drawing such diagrams often.

More generally, you can take the product between higher-dimensional tensors, following the same rules for shape compatibility as outlined earlier for the 2D case:

```
(a, b, c, d) • (d,) -> (a, b, c)
(a, b, c, d) • (d, e) -> (a, b, c, e)
```

And so on.

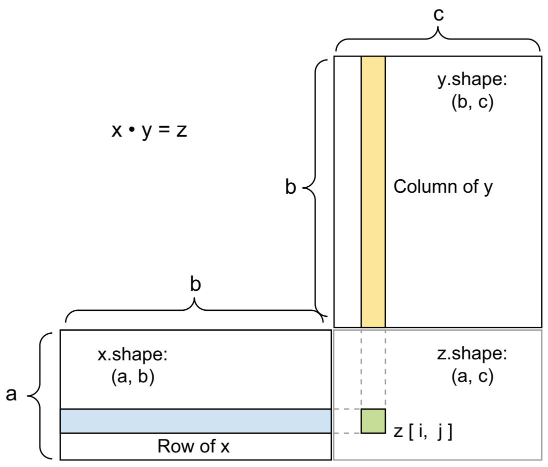

Figure 2.5 Matrix product box diagram

2.3.4 *Tensor reshaping*

A third type of tensor operation that's essential to understand is *tensor reshaping*. Although it wasn't used in the Dense layers in our first neural network example, we used it when we preprocessed the digits data before feeding it into our model:

```
train_images = train_images.reshape((60000, 28 * 28))
```

Reshaping a tensor means rearranging its rows and columns to match a target shape. Naturally, the reshaped tensor has the same total number of coefficients as the initial tensor. Reshaping is best understood via simple examples:

```
>>> x = np.array([[0., 1.],
...               [2., 3.],
...               [4., 5.]])
>>> x.shape
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

A special case of reshaping that's commonly encountered is *transposition*. *Transposing* a matrix means exchanging its rows and its columns, so that `x[i, :]` becomes `x[:, i]`:

```
>>> x = np.zeros((300, 20))      <-- Creates an all-zeros
>>> x = np.transpose(x)              matrix of shape (300, 20)
>>> x.shape
(20, 300)
```

2.3.5 Geometric interpretation of tensor operations

Because the contents of the tensors manipulated by tensor operations can be interpreted as coordinates of points in some geometric space, all tensor operations have a geometric interpretation. For instance, let's consider addition. We'll start with the following vector:

```
A = [0.5, 1]
```

It's a point in a 2D space (see figure 2.6). It's common to picture a vector as an arrow linking the origin to the point, as shown in figure 2.7.

Let's consider a new point, `B = [1, 0.25]`, which we'll add to the previous one. This is done geometrically by chaining together the vector arrows, with the resulting location being the vector representing the sum of the previous two vectors (see figure 2.8). As you can see, adding a vector `B` to a vector `A` represents the action of copying point A in

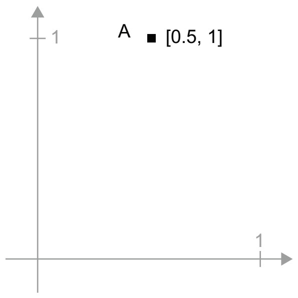

Figure 2.6 A point in a 2D space

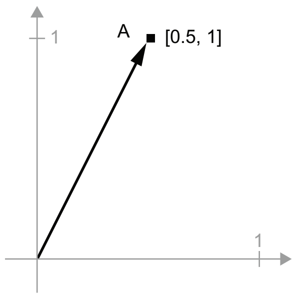

Figure 2.7 A point in a 2D space pictured as an arrow

new location, whose distance and direction from the original point A is determined by the vector B. If you apply the same vector addition to a group of points in the plane (an "object"), you would be creating a copy of the entire object in a new location (see figure 2.9). Tensor addition thus represents the action of *translating an object* (moving the object without distorting it) by a certain amount in a certain direction.

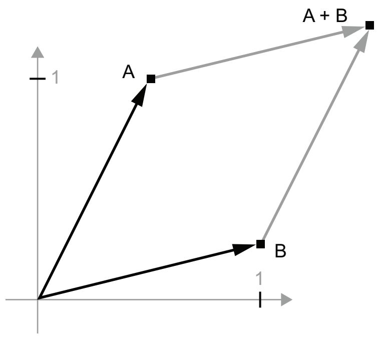

Figure 2.8 Geometric interpretation of the sum of two vectors

In general, elementary geometric operations, such as translation, rotation, scaling, skewing, and so on, can be expressed as tensor operations. Here are a few examples:

- *Translation*—As you just saw, adding a vector to a point will move this point by a fixed amount in a fixed direction. Applied to a set of points (such as a 2D object), this is called a "translation" (see figure 2.9).

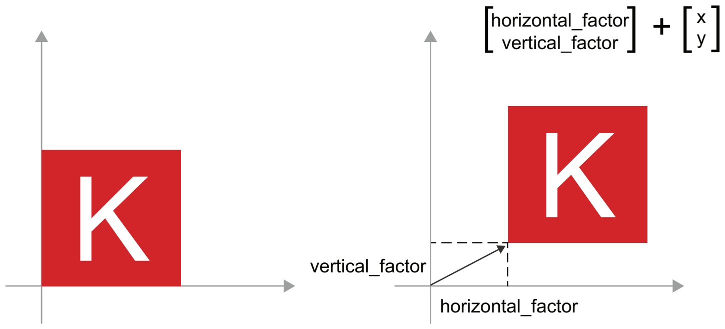

Figure 2.9 2D translation as a vector addition

- *Rotation*—A counterclockwise rotation of a 2D vector by an angle theta (see figure 2.10) can be achieved via a product with a 2 × 2 matrix `R = [[cos(theta), -sin(theta)], [sin(theta), cos(theta)]]`.

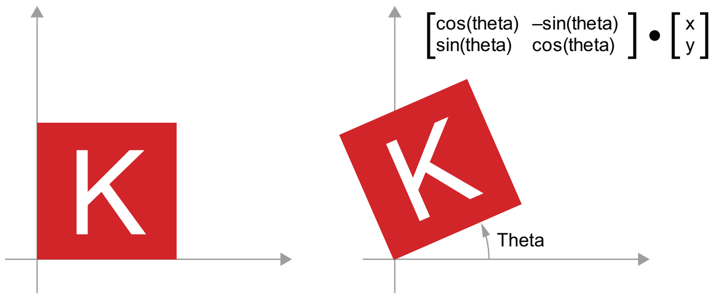

Figure 2.10 2D rotation (counterclockwise) as a matrix product

- *Scaling*—A vertical and horizontal scaling of the image (see figure 2.11) can be achieved via a product with a 2 × 2 matrix `S = ` (note that such a matrix is called a "diagonal matrix" because it only has non-zero coefficients in its "diagonal," going from the top left to the bottom right).

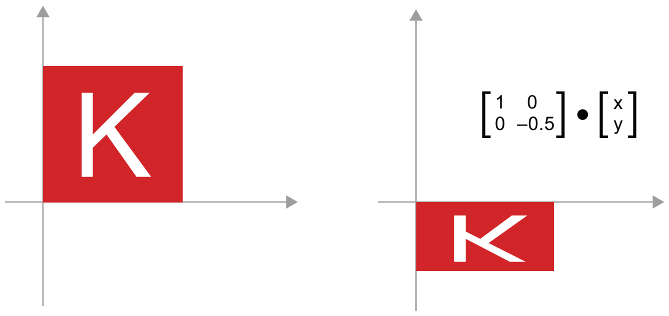

Figure 2.11 2D scaling as a matrix product

- *Linear transform*—A product with an arbitrary matrix implements a linear transform. Note that *scaling* and *rotation*, seen previously, are, by definition, linear transforms.
- *Affine transform*—An affine transform (see figure 2.12) is the combination of a linear transform (achieved via a matrix product) and a translation (achieved via a vector addition). As you have probably recognized, that's exactly the `y = W @ x + b` computation implemented by the `Dense` layer! A `Dense` layer without an activation function is an affine layer.

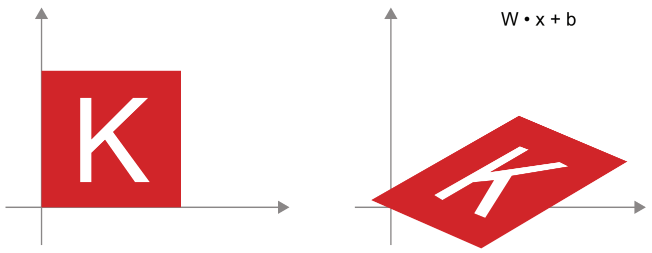

Figure 2.12 Affine transform in the plane

- *`Dense` layer with `relu` activation*—An important observation about affine transforms is that if you apply many of them repeatedly, you still end up with an affine transform (so you could just have applied that one affine transform in the first place). Let's try it with two: `affine2(affine1(x)) = W2 @ (W1 @ x + b1) + b2 = (W2 @ W1) @ x + (W2 @ b1 + b2)`. That's an affine transform where the linear part is the matrix `W2 @ W1` and the translation part is the vector `W2 @ b1 + b2`. As a consequence, a multilayer neural network made entirely of `Dense` layers without activations would be equivalent to a single `Dense` layer. This "deep" neural network would just be a linear model in disguise! This is why we need activation functions, like `relu` (seen in action in figure 2.13). Thanks to activation functions, a chain of `Dense` layers can be made to implement very complex, nonlinear geometric transformation, resulting in very rich hypothesis spaces for your deep neural networks. We cover this idea in more detail in the next chapter.

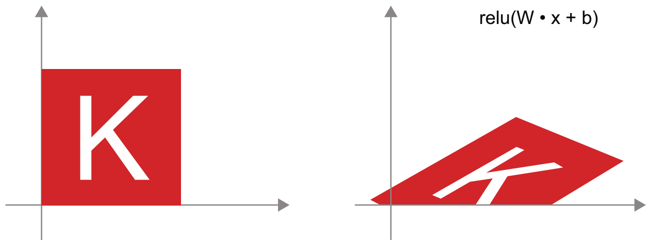

Figure 2.13 Affine transform followed by `relu` activation

2.3.6 A geometric interpretation of deep learning

You just learned that neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just simple geometric transformations of the input data. It follows that you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a series of simple steps.

In 3D, the following mental image may prove useful. Imagine two sheets of colored paper: one red and one blue. Put one on top of the other. Now crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem. What a neural network is meant to do is figure out a transformation of the paper ball that would uncrumple it to make the two classes cleanly separable again (see figure 2.14). With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.

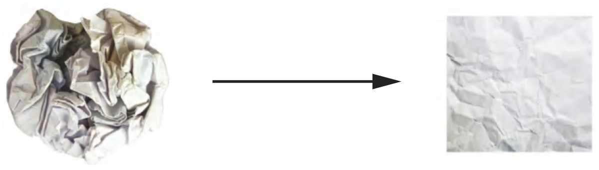

Figure 2.14 Uncrumpling a complicated manifold of data

Uncrumpling paper balls is what machine learning is about: finding neat representations for complex, highly folded data *manifolds* in high-dimensional spaces (a manifold is a continuous surface, like our crumpled sheet of paper). At this point, you should have a pretty good intuition as to why deep learning excels at this: it takes the approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much the strategy a human would follow to uncrumple a paper ball. Each layer in a deep network applies a transformation that disentangles the data a little—and a deep stack of layers makes tractable an extremely complicated disentanglement process.

2.4 The engine of neural networks: Gradient-based optimization

As you saw in the previous section, each neural layer from our first model example transforms its input data as follows:

```
output = relu(matmul(input, W) + b)
```

In this expression, `W` and `b` are tensors that are attributes of the layer. They're called the *weights* or *trainable parameters* of the layer (the `kernel` and `bias` attributes, respectively). These weights contain the information learned by the model from exposure to training data.

Initially, these weight matrices are filled with small random values (a step called *random initialization*). Of course, there's no reason to expect that `relu(matmul(input, W) + b)`, when `W` and `b` are random, will yield any useful representations. The resulting representations are meaningless—but they're a starting point. What comes next is to gradually adjust these weights, based on a feedback signal. This gradual adjustment, also called *training*, is basically the learning that machine learning is all about.

This happens within what's called a *training loop*, which works as follows. Repeat these steps in a loop, until the loss seems sufficiently low:

1. Draw a batch of training samples `x` and corresponding targets `y_true`.
2. Run the model on `x` (a step called the *forward pass*) to obtain predictions `y_pred`.
3. Compute the loss of the model on the batch, a measure of the mismatch between `y_pred` and `y_true`.
4. Update all weights of the model in a way that slightly reduces the loss on this batch.

You'll eventually end up with a model that has a very low loss on its training data: a low mismatch between predictions `y_pred` and expected targets `y_true`. The model has "learned" to map its inputs to correct targets. From afar, it may look like magic, but when you reduce it to elementary steps, it turns out to be simple.

Step 1 sounds easy enough—it's just I/O code. Steps 2 and 3 are merely the application of a handful of tensor operations, so you could implement these steps purely from what you learned in the previous section. The difficult part is step 4: updating the model's weights. Given an individual weight coefficient in the model, how can you compute whether the coefficient should be increased or decreased, and by how much?

One naive solution would be to freeze all weights in the model except the one scalar coefficient being considered and try different values for this coefficient. Let's say the initial value of the coefficient is 0.3. After the forward pass on a batch of data, the loss of the model on the batch is 0.5. If you change the coefficient's value to 0.35 and rerun the forward pass, the loss increases to 0.6. But if you lower the coefficient to 0.25, the loss falls to 0.4. In this case, it seems that updating the coefficient by –0.05 would contribute to minimizing the loss. This would have to be repeated for all coefficients in the model.

But such an approach would be horribly inefficient because you'd need to compute two forward passes (which are expensive) for every individual coefficient (of which there are many, usually at least a few thousands and potentially up to billions). Thankfully, there's a much better approach: *gradient descent.*

Gradient descent is the optimization technique that powers modern neural networks. Here's the gist of it. All of the functions used in our models (such as `matmul` or `+`) transform their input in a smooth and continuous way: if you look at `z = x + y`, for instance, a small change in `y` only results in a small change in `z`, and if you know the direction of the change in `y`, you can infer the direction of the change in `z`. Mathematically, you'd say these functions are *differentiable.* If you chain together such functions, the bigger function you obtain is still differentiable. In particular, this applies to the function that maps the model's coefficients to the loss of the model on a batch of data: a small change of the model's coefficients results in a small, predictable change of the loss value. This enables you to use a mathematical operator called the *gradient* to describe how the loss varies as you move the model's coefficients in different directions. If you compute this gradient, you can use it to move the coefficients (all at once in a single update, rather than one at a time) in a direction that decreases the loss.

If you already know what *differentiable* means and what a *gradient* is, you can skip the next two sections. Otherwise, the following will help you understand these concepts.

2.4.1 What's a derivative?

Consider a continuous, smooth function `f(x) = y`, mapping a number `x` to a new number `y`. We can use the function in figure 2.15 as an example.

Because the function is *continuous*, a small change in `x` can only result in a small change in `y`—that's the intuition behind *continuity*. Let's say you increase `x` by a small factor `epsilon_x`: this results in a small `epsilon_y` change to `y`, as shown in figure 2.16.

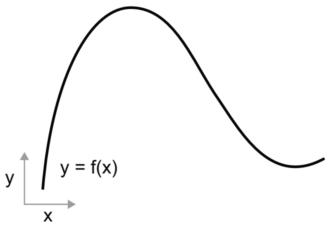

Figure 2.15 A continuous, smooth function

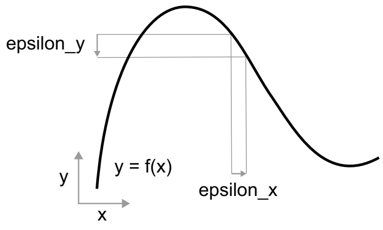

Figure 2.16 With a continuous function, a small change in `x` results in a small change in `y`.

In addition, because the function is *smooth* (its curve doesn't have any abrupt angles), when `epsilon_x` is small enough, around a certain point `p`, it's possible to approximate `f` as a linear function of slope `a`, so that `epsilon_y` becomes `a * epsilon_x`:

```
f(x + epsilon_x) = y + a * epsilon_x
```

Obviously, this linear approximation is valid only when `x` is close enough to `p`.

The slope `a` is called the *derivative* of `f` in `p`. If `a` is negative, it means a small increase in `x` around `p` will result in a decrease of `f(x)`, as shown in figure 2.17, and if `a` is positive, a small increase in `x` will result in an increase of `f(x)`. Further, the absolute value of `a` (the *magnitude* of the derivative) tells you how quickly this increase or decrease will happen.

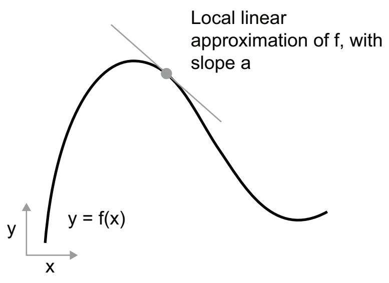

Figure 2.17 Derivative of `f` in `p`

For every differentiable function `f(x)` (*differentiable* means "can be derived": for example, smooth, continuous functions can be derived), there exists a derivative function `f'(x)` that maps values of `x` to the slope of the local linear approximation of `f` in those points. For instance, the derivative of `cos(x) is -sin(x)`, the derivative of `f(x) = a * x` is `f'(x) = a`, and so on.

Being able to derive functions is a very powerful tool when it comes to *optimization*, the task of finding values of `x` that minimize the value of `f(x)`. If you're trying to update `x` by a factor `epsilon_x` to minimize `f(x)` and you know the derivative of `f`, then your job is done: the derivative completely describes how `f(x)` evolves as you change `x`. If you want to reduce the value of `f(x)`, you just need to move `x` a little in the opposite direction from the derivative.

2.4.2 Derivative of a tensor operation: The gradient

The function we were just looking at turned a scalar value `x` into another scalar value `y`: you could plot it as a curve in a 2D plane. Now, imagine a function that turns a tuple of scalars `(x, y)` into a scalar value `z`: that would be a vector operation. You could plot it as a 2D *surface* in a 3D space (indexed by coordinates `x, y, z`). Likewise, you can imagine functions that take as input matrices, functions that take as input rank-3 tensors, etc.

The concept of derivation can be applied to any such function, as long as the surfaces they describe are continuous and smooth. The derivative of a tensor operation (or tensor function) is called a *gradient*. Gradients are just the generalization of the concept of derivatives to functions that take tensors as inputs. Remember how, for a scalar function, the derivative represents the *local slope* of the curve of the function? In just the same way, the gradient of a tensor function represents the *curvature* of the multidimensional surface described by the function. It characterizes how the output of the function varies when its input parameters vary.

Let's look at an example grounded in machine learning. Consider

- An input vector `x` (a sample in a dataset)
- A matrix `W` (the weights of a model)
- A target `y_true` (what the model should learn to associated to `x`)
- A loss function `loss` (meant to measure the gap between the model's current predictions and `y_true`).

You can use `W` to compute a target candidate `y_pred` and then compute the loss, or mismatch, between the target candidate `y_pred` and the target `y_true`:

```
y_pred = matmul(x, W)
loss_value = loss(y_pred, y_true)
```

We use the model weights W to make a prediction for x.

We estimate how far off the prediction was.

Now, we'd like to use gradients to figure out how to update `W` to make `loss_value` smaller. How do we do that?

Given fixed inputs `x` and `y_true`, the previous operations can be interpreted as a function mapping values of `W` (the model's weights) to loss values:

```
loss_value = f(W)
```

f describes the curve (or high-dimensional surface) formed by loss values when W varies.

Let's say the current value of `W` is `W0`. Then the derivative of `f` in the point `W0` is a tensor `grad(loss_value, W0)`, with the same shape as `W`, where each coefficient `grad(loss_value, W0)[i, j]` indicates the direction and magnitude of the change in `loss_value` you observe when modifying `W0[i, j]`. That tensor `grad(loss_value, W0)` is the gradient of the function `f(W) = loss_value` in `W0`, also called "gradient of `loss_value` with respect to `W` around `W0`."

NOTE The tensor operation `grad(f(W), W)` (which takes as input a matrix `W`) can be expressed as a combination of scalar functions `grad_ij(f(W), w_ij)`, each of which would return the derivative of `loss_value = f(W)` with respect to the coefficient `W[i, j]` of `W`, assuming all other coefficients are constant. `grad_ij` is called the *partial derivative* of `f` with respect to `W[i, j]`.

Concretely, what does `grad(loss_value, W0)` represent? You saw earlier that the derivative of a function `f(x)` of a single coefficient can be interpreted as the slope of the curve of `f`. Likewise, `grad(loss_value, W0)` can be interpreted as the tensor describing the *curvature* of `loss_value = f(W)` around `W0`. Each partial derivative describes the curvature of `f` in a specific direction.

We just saw how for a function `f(x)`, you can reduce the value of `f(x)` by moving `x` a little in the opposite direction from the derivative. In much the same way, with a function `f(W)` of a tensor, you can reduce `loss_value = f(W)` by moving `W` in the opposite direction from the gradient, such as an update of `W1 = W0 - step * grad(f(W0), W0)` where `step` is a small scaling factor. That means going against the curvature, which intuitively should put you lower on the curve. Note that the scaling factor `step` is needed because `grad(loss_value, W0)` only approximates the curvature when you're close to `W0`, so you don't want to get too far from `W0`.

2.4.3 Stochastic gradient descent

Given a differentiable function, it's theoretically possible to find its minimum analytically: it's known that a function's minimum is a point where the derivative is 0, so all you have to do is find all the points where the derivative goes to 0 and check for which of these points the function has the lowest value.

Applied to a neural network, that means finding analytically the combination of weight values that yields the smallest possible loss function. This can be done by solving the equation `grad(f(W), W) = 0` for `W`. This is a polynomial equation of `N` variables, where `N` is the number of coefficients in the model. Although it would be possible to solve such an equation for `N = 2` or `N = 3`, doing so is intractable for real neural networks, where the number of parameters is never less than a few thousand and can sometimes be in the billions.

Instead, you can use the four-step algorithm outlined at the beginning of this section: modify the parameters little by little based on the current loss value on a random batch of data. Because you're dealing with a differentiable function, you can compute its gradient, which gives you an efficient way to implement step 4. If you update the weights in the opposite direction from the gradient, the loss will be a little less every time:

1. Draw a batch of training samples `x` and corresponding targets `y_true`.
2. Run the model on `x` to obtain predictions `y_pred` (this is called the *forward pass*).
3. Compute the loss of the model on the batch, a measure of the mismatch between `y_pred` and `y_true`.
4. Compute the gradient of the loss with regard to the model's parameters (this is called the *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example, `W -= learning_rate * gradient`—thus reducing the loss on the batch a bit. The *learning rate* (`learning_rate` here) would be a scalar factor modulating the "speed" of the gradient descent process.

Easy enough! What we just described is called *mini-batch stochastic gradient descent* (mini-batch SGD). The term *stochastic* refers to the fact that each batch of data is drawn at random (*stochastic* is a scientific synonym of *random*). Figure 2.18 illustrates what happens in 1D, when the model has only one parameter and you have only one training sample.

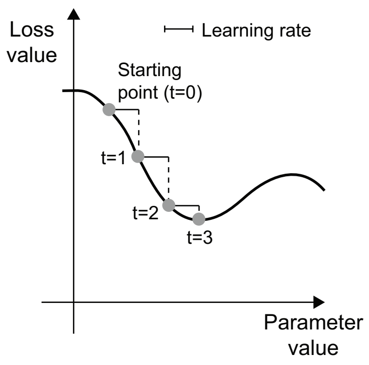

Figure 2.18 SGD down a 1D loss curve (one learnable parameter)

We can see intuitively that it's important to pick a reasonable value for the `learning_rate` factor. If it's too small, the descent down the curve will take many iterations, and it could get stuck in a local minimum. If `learning_rate` is too large, your updates may end up taking you to completely random locations on the curve.

Note that a variant of the mini-batch SGD algorithm would be to draw a single sample and target at each iteration, rather than drawing a batch of data. This would be *true* SGD (as opposed to *mini-batch* SGD). Alternatively, going to the opposite extreme, you could run every step on *all* data available, which is called *batch gradient descent*. Each update would then be more accurate, but far more expensive. The efficient compromise between these two extremes is to use mini-batches of reasonable size.

Although figure 2.18 illustrates gradient descent in a 1D parameter space, in practice, you'll use gradient descent in highly dimensional spaces: every weight coefficient in a neural network is a free dimension in the space, and there may be tens of thousands or even millions of them. To help you build intuition about loss surfaces, you can also visualize gradient descent along a 2D loss surface, as shown in figure 2.19. But you can't possibly visualize what the actual process of training a neural network looks like—you can't represent a 1,000,000-dimensional space in a way that makes sense to humans. As such, it's good to keep in mind that the intuitions you develop through these low-dimensional representations may not always be accurate in practice. This has historically been a source of issues in the world of deep learning research.

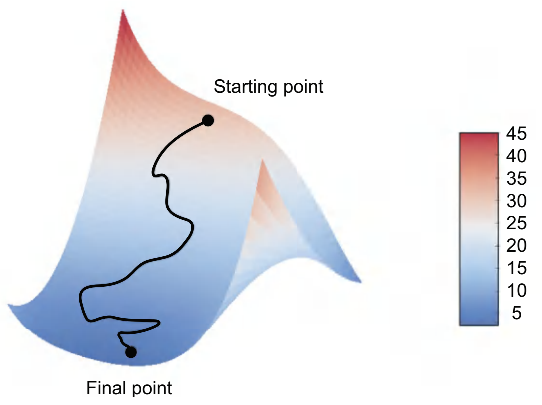

Figure 2.19 Gradient descent down a 2D loss surface (two learnable parameters)

Additionally, there exist multiple variants of SGD that differ by taking into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients. There is, for instance, SGD with momentum, as well as Adagrad, RMSprop, and several others. Such variants are known as *optimization methods* or *optimizers*. In particular, the concept of *momentum*, which is used in many of these variants, deserves your attention. Momentum addresses two issues with SGD: convergence speed and local minima. Consider figure 2.20, which shows the curve of a loss as a function of a model parameter.

As you can see, around a certain parameter value, there is a *local minimum*: around that point, moving left would result in the loss increasing, but so would moving right. If the parameter under consideration were being optimized via SGD with a small learning rate, then the optimization process would get stuck at the local minimum instead of making its way to the global minimum.

You can avoid such issues by using momentum, which draws inspiration from physics. A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve. If it has enough

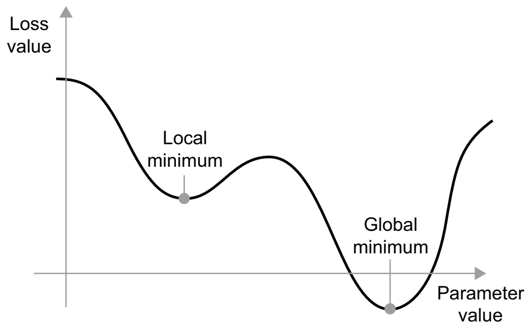

Figure 2.20 A local minimum and a global minimum

momentum, the ball won't get stuck in a ravine and will end up at the global minimum. Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration). In practice, this means updating the parameter `w` based not only on the current gradient value but also on the previous parameter update, such as in this naive implementation:

```
past_velocity = 0.0
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

Constant momentum factor

Optimization loop

2.4.4 *Chaining derivatives: The Backpropagation algorithm*

In the previously discussed algorithm, we casually assumed that because a function is differentiable, we can easily compute its gradient. But is that true? How can we compute the gradient of complex expressions in practice? In our two-layer network example, how can we get the gradient of the loss with regard to the weights? That's where the *Backpropagation algorithm* comes in.

THE CHAIN RULE

Backpropagation is a way to use the derivative of simple operations (such as addition, `relu`, or tensor product) to easily compute the gradient of arbitrarily complex combinations of these atomic operations. Crucially, a neural network consists of many tensor operations chained together, each of which has a simple, known derivative. For instance, the model from our first example can be expressed as a function parameterized by the variables `W1`, `b1`, `W2`, and `b2` (belonging to the first and second `Dense` layers, respectively), involving the atomic operations `matmul`, `relu`, `softmax`, and `+`, as well as our loss function, `loss`, which are all easily differentiable:

```
loss_value = loss(
    y_true,
    softmax(matmul(relu(matmul(inputs, W1) + b1), W2) + b2),
)
```

Calculus tells us that such a chain of functions can be derived using the following identity, called the *chain rule*. Consider two functions `f` and `g`, as well as the composed function `fg` such that `y = fg(x) == f(g(x))`:

```
def fg(x):
    x1 = g(x)
    y = f(x1)
    return y
```

Then the chain rule states that `grad(y, x) == grad(y, x1) * grad(x1, x)`. This enables you to compute the derivative of `fg` as long as you know the derivatives of `f` and `g`. The chain rule is named like this because when you add more intermediate functions, it starts looking like a chain:

```
def fghj(x):
    x1 = j(x)
    x2 = h(x1)
    x3 = g(x2)
    y = f(x3)
    return y

grad(y, x) == grad(y, x3) * grad(x3, x2) * grad(x2, x1) * grad(x1, x)
```

Applying the chain rule to the computation of the gradient values of a neural network gives rise to an algorithm called *backpropagation*. Let's see how that works, concretely.

AUTOMATIC DIFFERENTIATION WITH COMPUTATION GRAPHS

A useful way to think about backpropagation is in terms of *computation graphs*. A computation graph is the data structure at the heart of the deep learning revolution. It's a directed acyclic graph of operations—in our case, tensor operations. For instance, figure 2.21 is the graph representation of our first model.

Computation graphs have been an extremely successful abstraction in computer science because they enable us to *treat computation as data*: a computable expression is encoded as a machine-readable data structure that can be used as the input or output of another program. For instance, you could imagine a program that receives a computation graph and returns a new computation graph that implements a large-scale distributed version of the same computation—this would mean that you could distribute any computation without having to write the distribution logic yourself. Or imagine … a program that receives a computation graph and can automatically generate the derivative of the expression it represents. It's much easier to do these things if your computation is expressed as an explicit graph data structure rather than, say, lines of ASCII characters in a `.py` file.

To explain backpropagation clearly, let's look at a really basic example of a computation graph. We'll consider a simplified version of the graph in figure 2.21, where we only have one linear layer and where all variables are scalar, shown in figure 2.22. We'll take two scalar variables `w`, `b`, a scalar input `x`, and apply some operations to them to combine into an output `y`. Finally, we'll apply an absolute value error loss function: `loss_val = abs(y_true - y)`. Since we want to update `w` and `b` in a way that would minimize `loss_val`, we are interested in computing `grad(loss_val, b)` and `grad(loss_val, w)`.

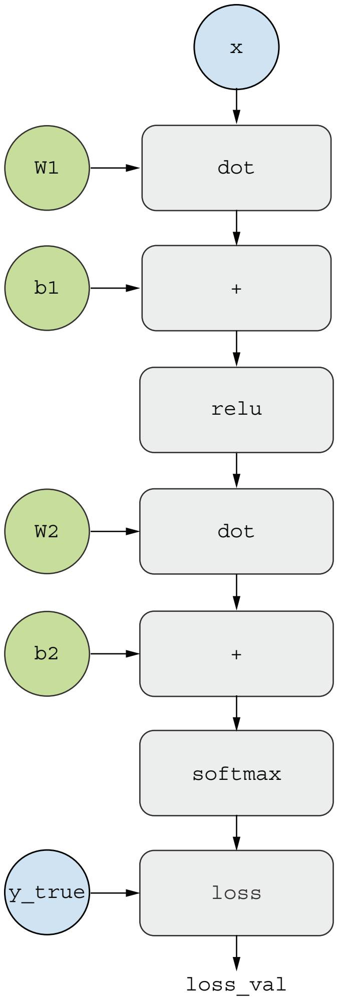

Figure 2.21 The computation graph representation of our two-layer model

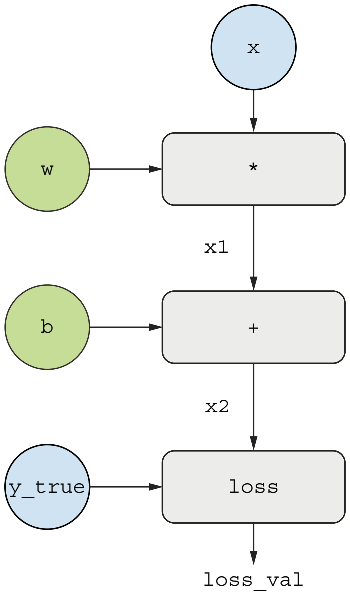

Figure 2.22 A basic example of a computation graph

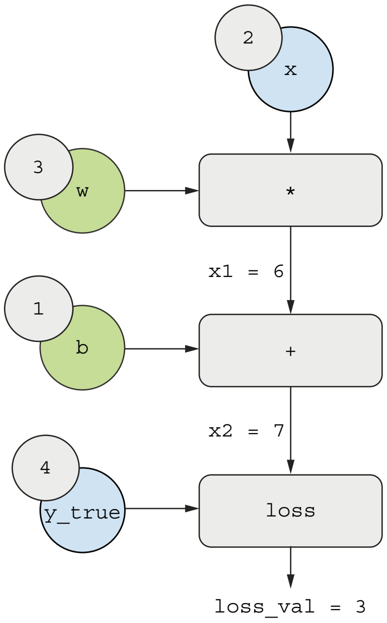

Figure 2.23 Running a forward pass

Let's set concrete values for the "input nodes" in the graph—that is, the input `x`, the target `y_true`, `w` and `b` (figure 2.23). We propagate these values to all nodes in the graph, from top to bottom, until we reach `loss_val`. This is the *forward pass.*

Now let's "reverse" the graph: for each edge in the graph going from `A` to `B`, we will create an opposite edge from `B` to `A`, and ask, "How much does `B` vary when `A` varies?" That is, what is grad(B, A)? We'll annotate each inverted edge with this value (figure 2.24). This backward graph represents the *backward pass*.

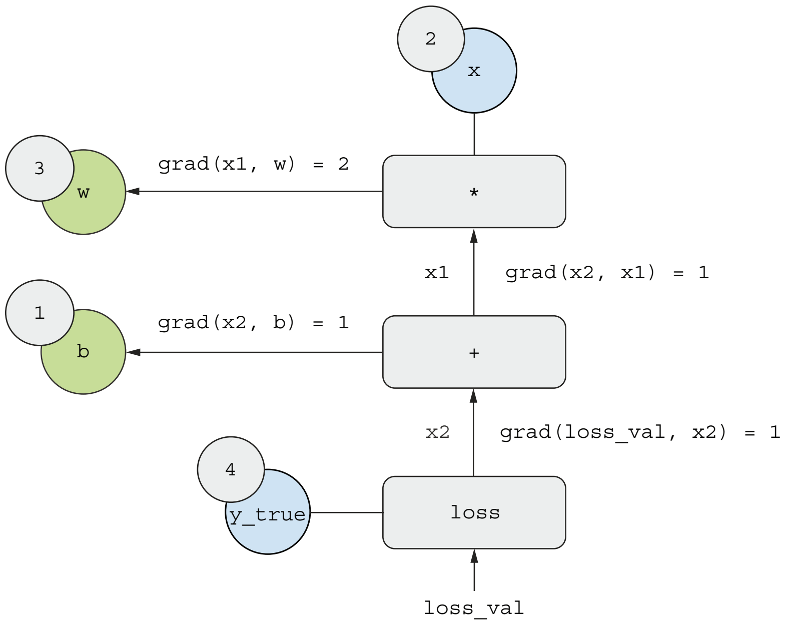

Figure 2.24 Running a backward pass

We have

- grad(loss_val, x2) = 1 because as x2 varies by an amount epsilon, loss_val = abs(4 - x2) varies by the same amount.
- grad(x2, x1) = 1 because as x1 varies by an amount epsilon, x2 = x1 + b = x1 + 1 varies by the same amount.
- grad(x2, b) = 1 because as b varies by an amount epsilon, x2 = x1 + b = 6 + b varies by the same amount.
- grad(x1, w) = 2 because as w varies by an amount epsilon, x1 = x * w = 2 * w varies by 2 * epsilon.

What the chain rule says about this backward graph is that you can obtain the derivative of a node with respect to another node by *multiplying the derivatives for each edge along the path linking the two nodes*. For instance, grad(loss_val, w) = grad(loss_val, x2) * grad(x2, x1) * grad(x1, w) (see figure 2.25).

By applying the chain rule to our graph, we obtain what we were looking for:

- grad(loss_val, w) = 1 * 1 * 2 = 2
- grad(loss_val, b) = 1 * 1 = 1

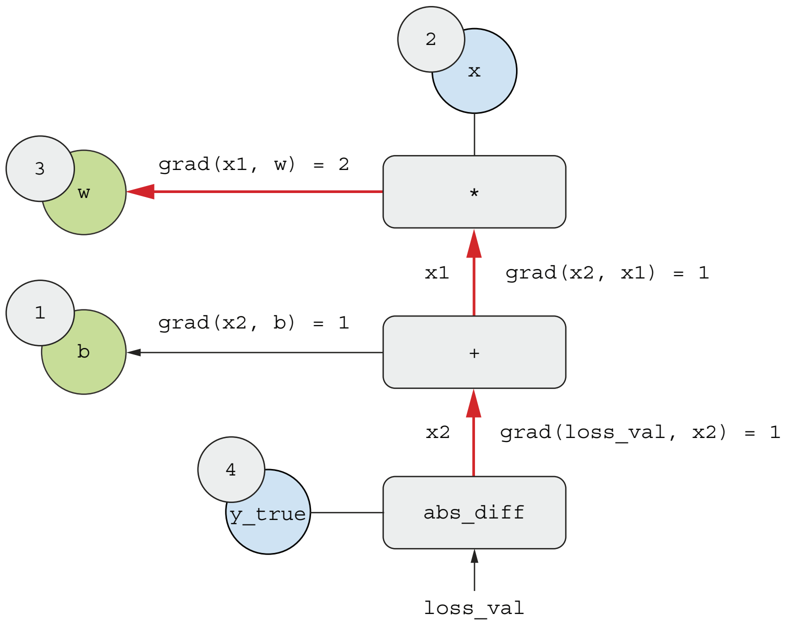

Figure 2.25 Path from `loss_val` to `w` in the backward graph

> **NOTE** If there are multiple paths linking the two nodes of interest `a`, `b` in the backward graph, we would obtain `grad(b, a)` by summing the contributions of all the paths.

And with that, you just saw backpropagation in action! Backpropagation is simply the application of the chain rule to a computation graph. There's nothing more to it. Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, computing the contribution that each parameter had in the loss value. That's where the name "backpropagation" comes from: we "back propagate" the loss contributions of different nodes in a computation graph.

Nowadays, people implement neural networks in modern frameworks that are capable of *automatic differentiation*, such as JAX, TensorFlow, and PyTorch. Automatic differentiation is implemented with the kind of computation graph previously presented. Automatic differentiation makes it possible to retrieve the gradients of arbitrary compositions of differentiable tensor operations without doing any extra work besides writing down the forward pass. When I wrote my first neural networks in C in the 2000s, I had to write my gradients by hand. Now, thanks to modern automatic differentiation tools, you'll never have to implement backpropagation yourself. Consider yourself lucky!

2.5 Looking back at our first example

You're nearing the end of this chapter, and you should now have a general understanding of what's going on behind the scenes in a neural network. What was a magical black box at the start of the chapter has turned into a clearer picture, as illustrated in figure 2.26: the model, composed of layers that are chained together, maps the input data to predictions. The loss function then compares these predictions to the targets, producing a loss value: a measure of how well the model's predictions match what was expected. The optimizer uses this loss value to update the model's weights.

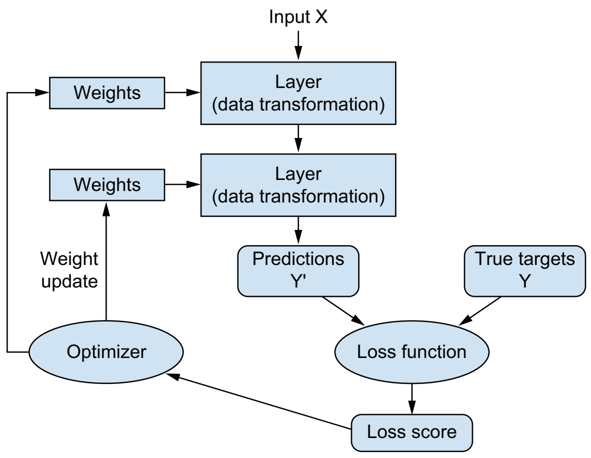

Figure 2.26 Relationship between the network, layers, loss function, and optimizer

Let's go back to the first example and review each piece of it in the light of what you've learned in the previous sections.

This was the input data:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

Now you understand that the input images are stored in NumPy tensors, which are here formatted as `float32` tensors of shape `(60000, 784)` (training data) and `(10000, 784)` (test data), respectively.

This was our model:

```
model = keras.Sequential(
    [
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax"),
    ]
)
```

Now you understand that this model consists of a chain of two `Dense` layers, that each layer applies a few simple tensor operations to the input data, and that these operations involve weight tensors. Weight tensors, which are attributes of the layers, are where the *knowledge* of the model persists.

This was the model-compilation step:

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

Now you understand that `"sparse_categorical_crossentropy"` is the loss function that's used as a feedback signal for learning the weight tensors, which the training phase will attempt to minimize. You also know that this reduction of the loss happens via mini-batch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the `"adam"` optimizer passed as the first argument.

Finally, this was the training loop:

```
model.fit(
    train_images,
    train_labels,
    epochs=5,
    batch_size=128,
)
```

Now you understand what happens when you call `fit`: the model will start to iterate on the training data in mini-batches of 128 samples, 5 times over (each iteration over all the training data is called an *epoch*). For each batch, the model will compute the gradient of the loss with regard to the weights (using the Backpropagation algorithm, which derives from the chain rule in calculus) and move the weights in the direction that will reduce the value of the loss for this batch.

After these 5 epochs, the model will have performed 2,345 gradient updates (469 per epoch), and the loss of the model will be sufficiently low that the model will be capable of classifying handwritten digits with high accuracy.

At this point, you already know most of what there is to know about neural networks. Let's prove it by reimplementing a simplified version of that first example step by step, using only low-level operations.

2.5.1 Reimplementing our first example from scratch

What's better to demonstrate full, unambiguous understanding than to implement everything from scratch? Of course, what "from scratch" means here is relative: we won't reimplement basic tensor operations, and we won't implement backpropagation. But we'll go to such a low level that each computation step will be made explicit.

Don't worry if you don't understand every little detail in this example just yet. The next chapter will dive in more detail into the Keras API. For now, just try to follow the gist of what's going on—the intent of this example is to help crystallize your understanding of the mathematics of deep learning using a concrete implementation. Let's go!

A simple Dense class

You've learned earlier that the `Dense` layer implements the following input transformation, where `W` and `b` are model parameters, and `activation` is an element-wise function (usually `relu`):

```
output = activation(matmul(input, W) + b)
```

Let's implement a simple Python class `NaiveDense` that creates two Keras variables `W` and `b`, and exposes a `__call__()` method that applies the previous transformation:

```
import keras
from keras import ops

class NaiveDense:
    def __init__(self, input_size, output_size, activation=None):
        self.activation = activation
        self.W = keras.Variable(
            shape=(input_size, output_size), initializer="uniform"
        )
        self.b = keras.Variable(shape=(output_size,), initializer="zeros")

    def __call__(self, inputs):
        x = ops.matmul(inputs, self.W)
        x = x + self.b
        if self.activation is not None:
            x = self.activation(x)
        return x
```

keras.ops is where you will find all the tensor operations you need.

Creates a matrix W of shape (input_size, output_size), initialized with random values drawn from a uniform distribution

Creates a vector b of shape (output_size,), initialized with zeros

Applies the forward pass

```
    @property
    def weights(self):
        return [self.W, self.b]
```

The convenience method for retrieving the layer's weights

A SIMPLE SEQUENTIAL CLASS

Now, let's create a `NaiveSequential` class to chain these layers. It wraps a list of layers and exposes a `__call__()` method that simply calls the underlying layers on the inputs, in order. It also features a `weights` property to easily keep track of the layers' parameters:

```
class NaiveSequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, inputs):
        x = inputs
        for layer in self.layers:
            x = layer(x)
        return x

    @property
    def weights(self):
        weights = []
        for layer in self.layers:
            weights += layer.weights
        return weights
```

Using this `NaiveDense` class and this `NaiveSequential` class, we can create a mock Keras model:

```
model = NaiveSequential(
    [
        NaiveDense(input_size=28 * 28, output_size=512, activation=ops.relu),
        NaiveDense(input_size=512, output_size=10, activation=ops.softmax),
    ]
)
assert len(model.weights) == 4
```

A BATCH GENERATOR

Next, we need a way to iterate over the MNIST data in mini-batches. This is easy:

```
import math

class BatchGenerator:
    def __init__(self, images, labels, batch_size=128):
        assert len(images) == len(labels)
        self.index = 0
        self.images = images
        self.labels = labels
        self.batch_size = batch_size
        self.num_batches = math.ceil(len(images) / batch_size)

    def next(self):
        images = self.images[self.index : self.index + self.batch_size]
        labels = self.labels[self.index : self.index + self.batch_size]
        self.index += self.batch_size
        return images, labels
```

2.5.2 *Running one training step*

The most difficult part of the process is the "training step": updating the weights of the model after running it on one batch of data. We need to

- Compute the predictions of the model for the images in the batch
- Compute the loss value for these predictions given the actual labels
- Compute the gradient of the loss with regard to the model's weights
- Move the weights by a small amount in the direction opposite to the gradient

Listing 2.9 A single step of training

```
def one_training_step(model, images_batch, labels_batch):
    predictions = model(images_batch)
    loss = ops.sparse_categorical_crossentropy(labels_batch, predictions)
    average_loss = ops.mean(loss)
    gradients = get_gradients_of_loss_wrt_weights(loss, model.weights)
    update_weights(gradients, model.weights)
    return loss
```

Runs the "forward pass"

Computes the gradient of the loss with regard to the weights. The output, gradients, is a list where each entry corresponds to a weight from the model. weights list. We haven't defined this function yet!

Updates the weights using the gradients. We haven't defined this function yet!

THE WEIGHT UPDATE STEP

As you already know, the purpose of the "weight update" step (represented by the `update_weights()` function) is to move the weights by "a bit" in a direction that will reduce the loss on this batch. The magnitude of the move is determined by the "learning rate," typically a small quantity. The simplest way to implement this `update_weights()` function is to subtract `gradient * learning_rate` from each weight:

```
learning_rate = 1e-3

def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        w.assign(w - g * learning_rate)
```

Assigns a new value to the variable, in place

In practice, you will almost never implement a weight update step like this by hand. Instead, you would use an `Optimizer` instance from Keras—like this:

```
from keras import optimizers

optimizer = optimizers.SGD(learning_rate=1e-3)

def update_weights(gradients, weights):
    optimizer.apply_gradients(zip(gradients, weights))
```

Gradient computation

Now, there's just one thing we're still missing: gradient computation (represented by the `get_gradients_of_loss_wrt_weights()` function in listing 2.9). In the previous section, we outlined how we could use the chain rule to obtain the gradients of a chain of functions given their individual derivatives, a process known as backpropagation. We could reimplement backpropagation from scratch here, but that would be rather cumbersome, especially since we're using a `softmax` operation and a crossentropy loss, which have fairly verbose derivatives.

Instead, we can rely on the automatic differentiation mechanism that's built into one of the low-level frameworks supported by Keras, such as TensorFlow, JAX, or PyTorch. For the sake of the example, let's go with TensorFlow here. You'll learn more about TensorFlow, JAX, and PyTorch in the next chapter.

The API through which you can use TensorFlow's automatic differentiation capabilities is the `tf.GradientTape` object. It's a Python scope that will "record" the tensor operations that run inside it, in the form of a computation graph (sometimes called a *tape*). This graph can then be used to retrieve the gradient of any scalar value with respect to any set of input values:

```
import tensorflow as tf

x = tf.zeros(shape=())
```

Instantiates a scalar tensor with value 0

```
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

Opens a GradientTape scope

Inside the scope, applies some tensor operations to our variable

Uses the tape to retrieve the gradient of the output y with respect to our variable x

Let's rewrite our function `one_training_step()` using the TensorFlow `GradientTape` (skipping the need for a separate `get_gradients_of_loss_wrt_weights()` function):

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        predictions = model(images_batch)
        loss = ops.sparse_categorical_crossentropy(labels_batch, predictions)
        average_loss = ops.mean(loss)
    gradients = tape.gradient(average_loss, model.weights)
    update_weights(gradients, model.weights)
    return average_loss
```

Now that our per-batch training step is ready, we can move on to implementing an entire epoch of training.

2.5.3 *The full training loop*

An epoch of training simply consists of the repetition of the training step for each batch in the training data, and the full training loop is simply the repetition of one epoch:

```
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"Epoch {epoch_counter}")
        batch_generator = BatchGenerator(images, labels)
        for batch_counter in range(batch_generator.num_batches):
            images_batch, labels_batch = batch_generator.next()
            loss = one_training_step(model, images_batch, labels_batch)
            if batch_counter % 100 == 0:
                print(f"loss at batch {batch_counter}: {loss:.2f}")
```

Let's test-drive it:

```
from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255

fit(model, train_images, train_labels, epochs=10, batch_size=128)
```

2.5.4 Evaluating the model

We can evaluate the model by taking the `argmax` of its predictions over the test images, and comparing it to the expected labels:

```
>>> predictions = model(test_images)
>>> predicted_labels = ops.argmax(predictions, axis=1)
>>> matches = predicted_labels == test_labels
>>> f"accuracy: {ops.mean(matches):.2f}"
accuracy: 0.83
```

All done! As you can see, it's quite a bit of work to do "by hand" what you can do in a few lines of Keras code. But because you've gone through these steps, you should now have a crystal-clear understanding of what goes on inside a neural network when you call `fit()`. Having this low-level mental model of what your code is doing behind the scenes will make you better able to take advantage of the high-level features of the Keras API.

2.6 Summary

- *Tensors* form the foundation of modern machine learning systems. They come in various flavors of `dtype`, `rank`, and `shape`.
- You can manipulate numerical tensors via *tensor operations* (such as addition, tensor product, or element-wise multiplication), which can be interpreted as encoding geometric transformations. In general, everything in deep learning is amenable to a geometric interpretation.
- Deep learning models consist of chains of simple tensor operations, parameterized by *weights*, which are themselves tensors. The weights of a model are where its "knowledge" is stored.
- *Learning* means finding a set of values for the model's weights that minimizes a *loss function* for a given set of training data samples and their corresponding targets.
- Learning happens by drawing random batches of data samples and their targets and computing the gradient of the model parameters with respect to the loss on the batch. The model parameters are then moved a bit (the magnitude of the

move is defined by the learning rate) in the opposite direction from the gradient. This is called *mini-batch gradient descent.*

- The entire learning process is made possible by the fact that all tensor operations in neural networks are differentiable, and thus it's possible to apply the chain rule of derivation to find the gradient function mapping the current parameters and current batch of data to a gradient value. This is called *backpropagation.*
- Two key concepts you'll see frequently in future chapters are *loss* and *optimizers.* These are the two things you need to define before you begin feeding data into a model:
  - The *loss* is the quantity you'll attempt to minimize during training, so it should represent a measure of success for the task you're trying to solve.
  - The *optimizer* specifies the exact way in which the gradient of the loss will be used to update parameters: for instance, it could be the RMSProp optimizer, SGD with momentum, and so on.

3 Introduction to TensorFlow, PyTorch, JAX, and Keras

This chapter covers

- A closer look at all major deep learning frameworks and their relationships
- An overview of how core deep learning concepts translate to code across all frameworks

This chapter is meant to give you everything you need to start doing deep learning in practice. First, you'll get familiar with three popular deep learning frameworks that can be used with Keras:

- TensorFlow (https://tensorflow.org)
- PyTorch (https://pytorch.org/)
- JAX (https://jax.readthedocs.io/)

Then, building on top of the first contact you've had with Keras in chapter 2, we'll review the core components of neural networks and how they translate to Keras APIs.

By the end of this chapter, you'll be ready to move on to practical, real-world applications—which will start with chapter 4.