天氣與人工智慧 II

Ch.2 The mathematical building blocks of neural networks

周哲維
david10188@gmail.com

2.1 A first look at a neural network

- A concrete example of a neural network that uses the Python library Keras to learn to classify handwritten digits.
- 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. 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.

2.1 A first look at a neural network

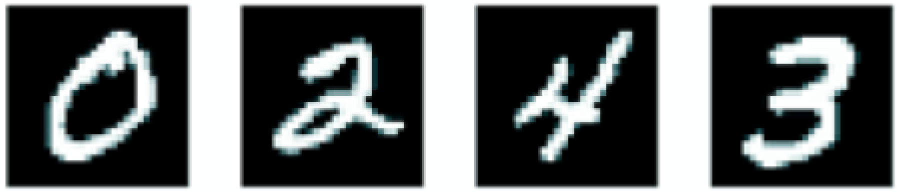


Figure 2.1 MNIST sample digits

2.1 *A first look at a neural network*

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.

2.1 A first look at a neural network

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)
```

2.1 A first look at a neural network

- 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`.

2.1 A first look at a neural network

- The core building block of neural networks is the *layer*, a data-processing module that you can think of as a filter for data. Some data goes in, and it comes out in a more useful form.

- Most of deep learning consists of chaining together simple layers that will implement a form of progressive *data distillation*.

2.1 A first look at a neural network

Listing 2.2 The network architecture

```
from keras import models
from keras import layers
network = models.Sequential()
network.add(layers.Dense(512, activation='relu',
input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

The network 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* layer, which means it will return an array of 10 probability scores (summing to 1).

2.1 A first look at a neural network

- To make the network ready for training, we need to pick three more things, as part of the *compilation* step:
  - *A loss function*—How the network 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 network will update itself based on the data it sees and its loss function.
  - *Metrics to monitor during training and testing*—Here, we’ll only care about accuracy (the fraction of the images that were correctly classified).

2.1 A first look at a neural network

Listing 2.3 The compilation step

```
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

2.1 A first look at a neural network

- Before training, we’ll preprocess the data by reshaping it into the shape the network expects and scaling it so that all values are in the [0, 1] interval.
- Previously, our training images, for instance, 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.
- We also need to categorically encode the labels, a step that’s explained in chapter 3.

2.1 A first look at a neural network

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
```

Listing 2.5 Preparing the labels

```
from keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

2.1 A first look at a neural network

- We’re now ready to train the network, which in Keras is done via a call to the network’s fit method—we *fit* the model to its training data:

```
>>> network.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [==============================] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [========================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

- Two quantities are displayed during training: the loss of the network over the training data, and the accuracy of the network over the training data.

2.1 A first look at a neural network

- We quickly reach an accuracy of 0.989 (98.9%) on the training data. Now let’s check that the model performs well on the test set, too:

```
>>> test_loss, test_acc = 
network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

- The test-set accuracy turns out to be 97.8%—that’s quite a bit lower than the training set 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 3.

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.
- So what’s a tensor?

2.2.1 Scalars (0D tensors)

- A tensor that contains only one number is called a *scalar* (or scalar tensor, or 0-dimensional tensor, or 0D tensor)
- 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 (1D tensors)

- An array of numbers is called a *vector*, or 1D tensor. A 1D tensor is said to have exactly one axis. Following is a Numpy vector:

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

2.2.3 Matrices (2D tensors)

- An array of vectors is a *matrix*, or 2D tensor. A matrix has two axes (often referred to *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 3D tensors and higher-dimensional tensors

- If you pack such matrices in a new array, you obtain a 3D tensor, which you can visually interpret as a cube of numbers. Following is a Numpy 3D 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
```

2.2.5 Key attributes

A tensor is defined by three key attributes:

- ***Number of axes (rank)***—For instance, a 3D tensor has three axes, and a matrix has two axes. This is also called the tensor's ndim in Python libraries such as Numpy.

- ***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 3D 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 float32, uint8, float64, and so on.

2.2.5 *Key attributes*

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, shape, dtype` attribute:

```
>>> print(train_images.ndim)
3
>>> print(train_images.shape)
(60000, 28, 28)
>>> print(train_images.dtype)
uint8
```

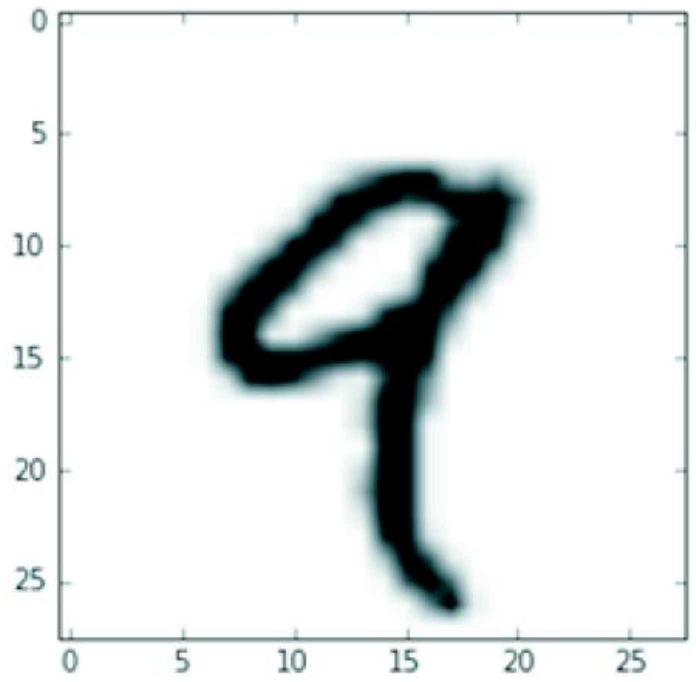
So what we have here is a 3D tensor of 8-bit integers. More precisely, it's an array of 60,000 matrices of 28 × 8 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255.

2.2.5 Key attributes

Let’s display the fourth digit in this 3D tensor, using the library Matplotlib

Listing 2.6 Displaying the fourth digit

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



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*.

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]
>>> print(my_slice.shape)
(90, 28, 28)
```

2.2.6 Manipulating tensors in Numpy

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)
```

2.2.6 Manipulating tensors in Numpy

In general, you may select between any two indices along each tensor axis. For instance, in order to select 14 × 14 pixels in the bottom-right corner of all images, you 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. In order to crop the images to patches of 14 × 14 pixels centered in the middle, you 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* (sometimes called the *samples dimension*). 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. Concretely, here’s one batch of our MNIST digits, with batch size of 128:

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

And here’s the next batch:

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

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:

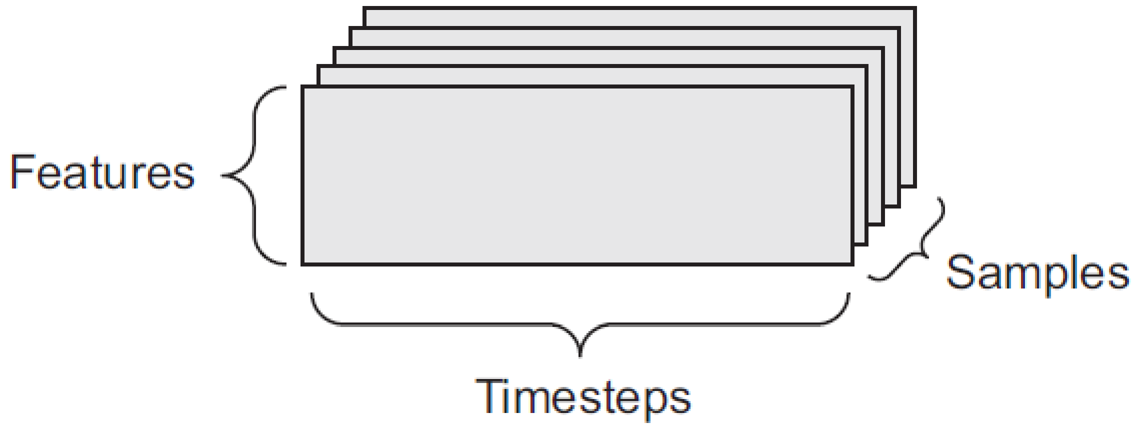
- *Vector data*—2D tensors of shape (`samples, features`)
- *Timeseries data or sequence data*—3D tensors of shape (`samples, timesteps, features`)
- *Images*—4D tensors of shape (`samples, height, width, channels`) or (`samples, channels, height, width`)
- *Video*—5D tensors of shape (`samples, frames, height, width, channels`) or (`samples, frames, channels, height, width`)

2.2.9 Vector data

This is the most common case. 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 2D tensor (that is, an array of vectors), where the first axis is the *samples axis* and the second axis is the *features axis*.

2.2.10 Timeseries data or sequence data

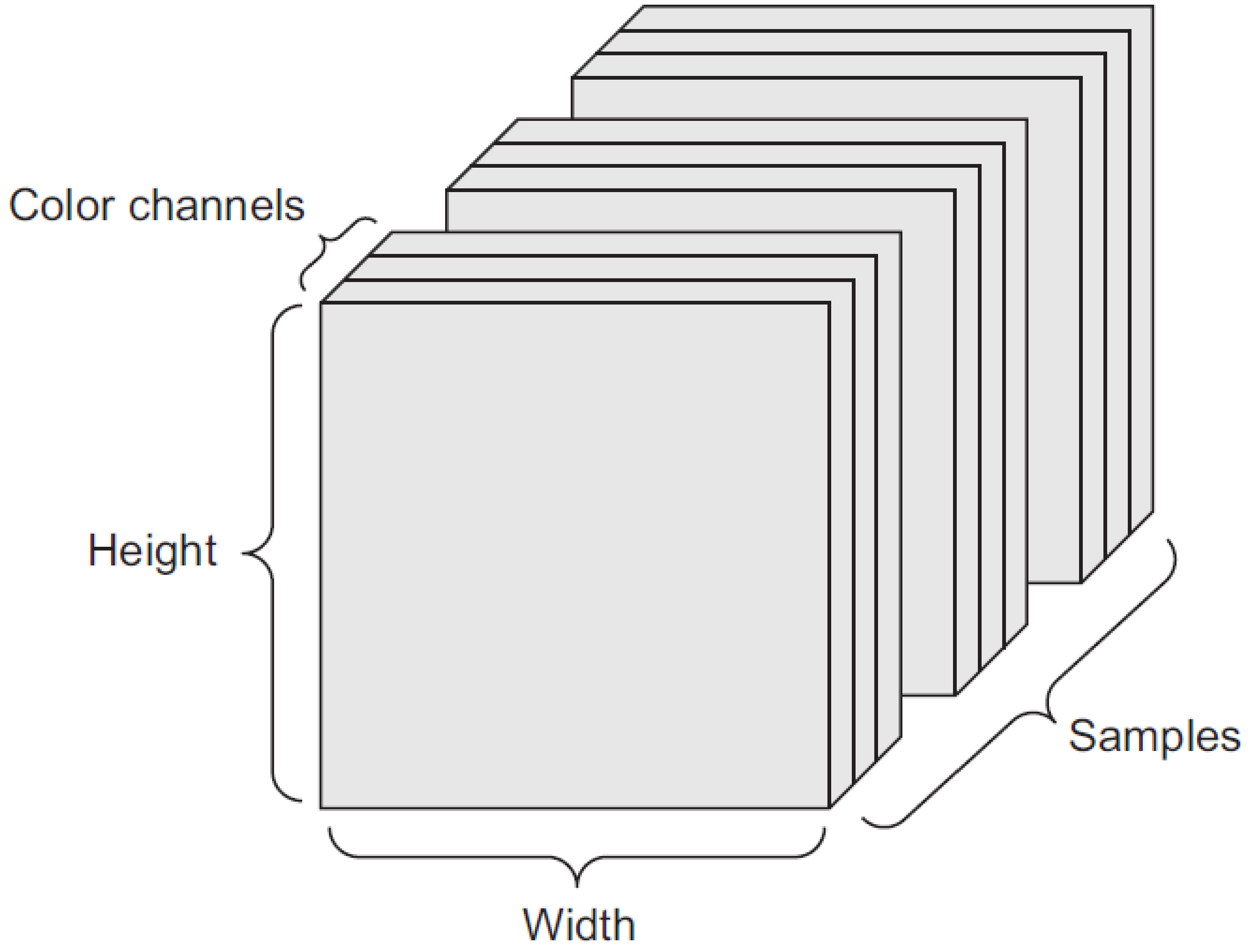
Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a 3D tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a 2D tensor), and thus a batch of data will be encoded as a 3D tensor.



2.2.11 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 2D tensors, by convention image tensors are always 3D, 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)`

2.2.11 Image data



2.2.12 Video data

- Video data is one of the few types of real-world data for which you’ll need 5D 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 3D tensor `(height, width, color_depth)`, a sequence of frames can be stored in a 4D tensor `(frames, height, width, color_depth)`, and thus a batch of different videos can be stored in a 5D tensor of shape `(samples, frames, height, width, color_depth)`.

2.2.12 Video data

- 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)`.

2.3 The gears of neural networks: tensor operations

In our initial example, we were building our network 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 2D tensor and returns another 2D tensor—a new representation for the input tensor. Specifically, the function is as follows (where `W` is a 2D tensor and `b` is a vector, both attributes of the layer):

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

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.

2.3.1 Element-wise operations

x is a 2D Numpy tensor.

```
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
```

Avoid overwriting the input tensor.

2.3.1 Element-wise operations

```
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 2D Numpy tensors.

Avoid overwriting the input tensor.

2.3.1 Element-wise operations

So, in Numpy, you can do the following element-wise operation:

```
import numpy as np
z = x + y
z = np.maximum(z, 0.)
```

Element-wise addition

Element-wise relu

2.3.2 Broadcasting

In the `Dense` layer introduced earlier, we added a 2D 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 *broadcasted* to match the shape of the larger tensor. Broadcasting consists of two steps:

1. Axes (called *broadcast axes*) are added to the smaller tensor to match the `ndim` of the larger tensor.
2. The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.

2.3.2 Broadcasting

Consider `X` with shape `(32, 10)` and `y` with shape `(10,)`.

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

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

2.3.2 Broadcasting

```
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 2D Numpy tensor.

y is a Numpy vector.

Avoid overwriting the input tensor.

2.3.2 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 dot

- The `dot` operation is the most common, most useful tensor operation.
- An element-wise product is done with the * operator in Numpy, Keras, Theano, and TensorFlow.
- `dot` uses a different syntax in TensorFlow, but in both Numpy and Keras it’s done using the standard dot operator:

```
import numpy as np
z = np.dot(x, y)
```

- In mathematical notation, you’d note the operation with a dot (.):

```
z = x . y
```

2.3.3 Tensor dot

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

x and y are Numpy vectors.

2.3.3 Tensor dot

```
import numpy as np
def naive_matrix_vector_dot(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 first dimension of x must be the same as the 0th dimension of y!

This operation returns a vector of 0s with the same shape as y.

2.3.3 Tensor dot

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

2.3.3 Tensor dot

The first dimension of x must be the same as the 0th dimension of y!

```
def naive_matrix_dot(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]
      z[i, j] = naive_vector_dot(row_x, column_y)
  return z
```

The first dimension of x must be the same as the 0th dimension of y!

2.3.3 Tensor dot

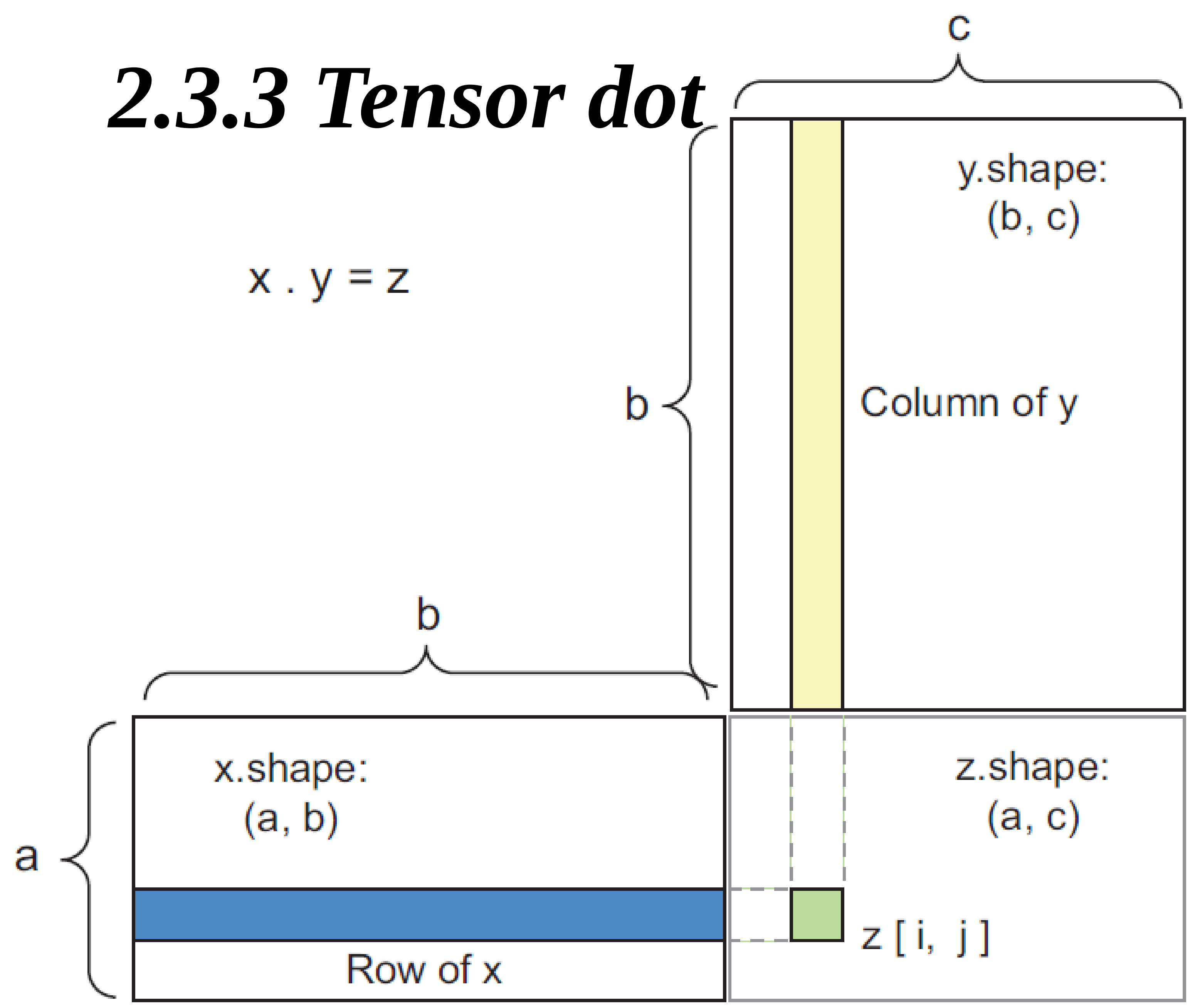


Figure 2.5 Matrix dot-product box diagram

2.3.3 Tensor dot

More generally, you can take the dot 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.

2.3.4 Tensor reshaping

We used *tensor reshaping* when we preprocessed the digits data before feeding it into our network:

```
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.

2.3.4 Tensor reshaping

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

```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> print(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.

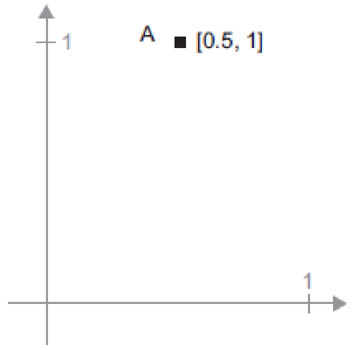


Figure 2.6 A point in a 2D space

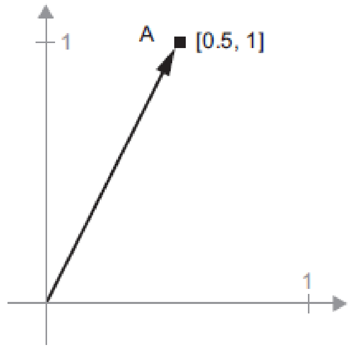


Figure 2.7 A point in a 2D space pictured as an arrow

2.3.5 Geometric interpretation of tensor operations

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).

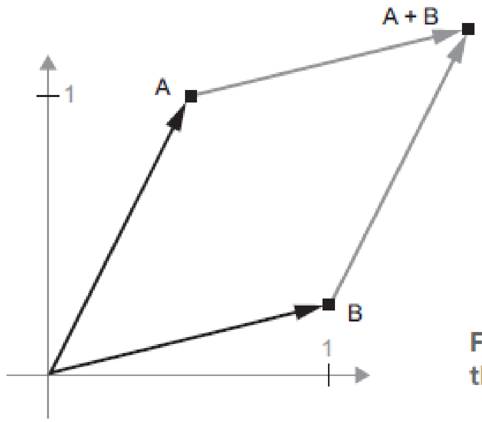


Figure 2.8 Geometric interpretation of the sum of two vectors

In general, elementary geometric operations such as affine transformations, rotations, scaling, and so on can be expressed as tensor operations. For instance, a rotation of a 2D vector by an angle theta can be achieved via a dot product with a 2 × 2 matrix `R = [u, v]`, where `u` and `v` are both vectors of the plane: `u = [cos(theta), sin(theta)]` and `v = [-sin(theta), cos(theta)]`.

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 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 long 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 (or any other machine-learning model) is meant to do is figure out a transformation of the paper ball that would uncrumple it, so as to make the two classes cleanly separable again. 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.

2.3.6 A geometric interpretation of deep learning

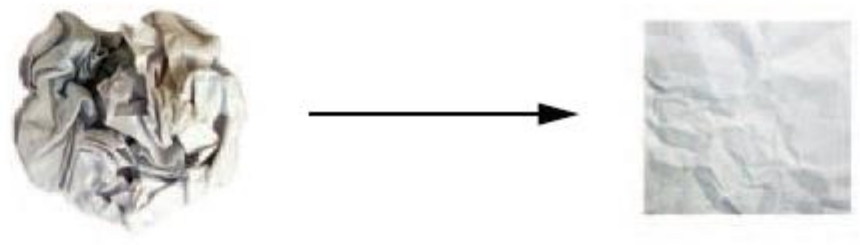


Figure 2.9 Uncrumpling a complicated manifold of data

Uncrumpling paper balls is what machine learning is about: finding neat representations for complex, highly folded data manifolds. 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

```
Output = relu(dot(W, input) + 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. These weights contain the information learned by the network from exposure to training data.

Initially, these weight matrices are filled with small random values (a step called *random initialization*). 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.

2.4 The engine of neural networks: gradient-based optimization

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

You’ll eventually end up with a network that has a very low loss on its training data: a low mismatch between predictions y_pred and expected targets y. The network has “learned” to map its inputs to correct targets.

2.4 The engine of neural networks: gradient-based optimization

- Step 1 sounds easy enough—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 network’s weights. Given an individual weight coefficient in the network, how can you compute whether the coefficient should be increased or decreased, and by how much?

2.4 The engine of neural networks: gradient-based optimization

- One naive solution would be to freeze all weights in the network 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 network 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 network.

2.4 The engine of neural networks: gradient-based optimization

- 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 thousands and sometimes up to millions).
- A much better approach is to take advantage of the fact that all operations used in the network are *differentiable*, and compute the *gradient* of the loss with regard to the network’s coefficients.
- You can then move the coefficients in the opposite direction from the gradient, thus decreasing the loss.

2.4.1 What's a derivative?

Consider a continuous, smooth function `f(x) = y`, mapping a real number `x` to a new real number `y`. Because the function is *continuous*, a small change in `x` can only result in a small change in `y`. Let's say you increase `x` by a small factor `epsilon_x`: this results in a small `epsilon_y` change to `y`:

```
f(x + epsilon_x) = y + epsilon_y
```

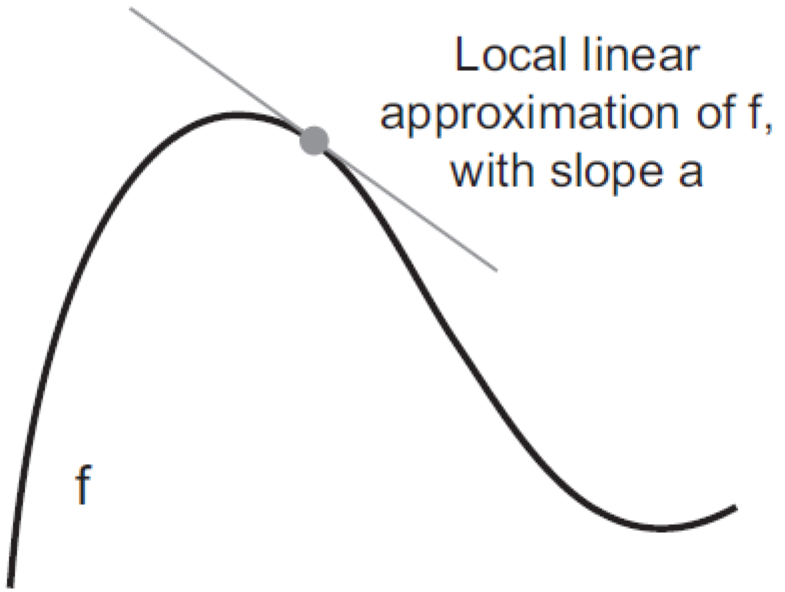
In addition, because the function is *smooth* (its curve doesn't have any abrupt angles), when `epsilon_x` is small enough, around a certain point `p`, it's possible to approximate `f` as a linear function of slope `a`, so that `epsilon_y` becomes `a * epsilon_x`:

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

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

2.4.1 What's a derivative?

The slope `a` is called the *derivative* of `f` in `p`. If `a` is negative, it means a small change of `x` around `p` will result in a decrease of `f(x)`; and if `a` is positive, a small change 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.



2.4.1 *What’s a derivative?*

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.

If you’re trying to update `x` by a factor `epsilon_x` in order 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

A *gradient* is the derivative of a tensor operation. It's the generalization of the concept of derivatives to functions of multidimensional inputs: that is, to functions that take tensors as inputs.

Consider an input vector `x`, a matrix `W`, a target `y`, and a loss function `loss`. You can use `W` to compute a target candidate `y_pred`, and compute the loss, or mismatch, between the target candidate `y_pred` and the target `y`:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y)
```

If the data inputs `x` and `y` are frozen, then this can be interpreted as a function mapping values of `W` to loss values:

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

2.4.2 Derivative of a tensor operation: the gradient

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

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, `gradient(f)(W0)` can be interpreted as the tensor describing the *curvature* of `f(W)` around `W0`.

2.4.2 Derivative of a tensor operation: the gradient

For this reason, in much the same way that, 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, with a function `f(W)` of a tensor, you can reduce `f(W)` by moving `W` in the opposite direction from the gradient: for example, `W1 = W0 - step * gradient(f)(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 `gradient(f)(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 `gradient(f)(W) = 0` for `W`. This is a polynomial equation of *N* variables, where *N* is the number of coefficients in the network. 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 often be several tens of millions.

2.4.3 Stochastic gradient descent

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`.
2. Run the network on `x` to obtain predictions `y_pred`.
3. Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Compute the gradient of the loss with regard to the network's parameters (a *backward pass*).
5. Move the parameters a little in the opposite direction from the gradient—for example `W -= step * gradient`—thus reducing the loss on the batch a bit.

2.4.3 Stochastic gradient descent

Easy enough! What I just described is called *mini-batch stochastic gradient descent* (minibatch 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 2.11 illustrates what happens in 1D, when the network has only one parameter and you have only one training sample.

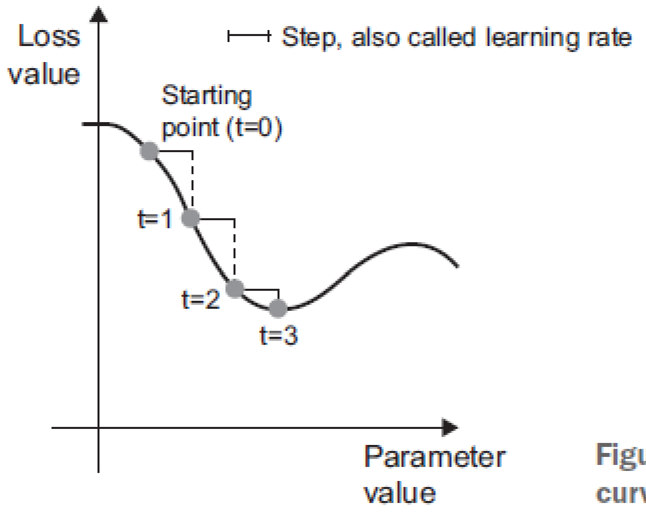


Figure 2.11 SGD down a 1D loss curve (one learnable parameter)

2.4.3 Stochastic gradient descent

As you can see, intuitively it’s important to pick a reasonable value for the `step` 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 `step` 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 SGD*. 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.

2.4.3 Stochastic gradient descent

Although figure 2.11 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 2.12. 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.

2.4.3 Stochastic gradient descent

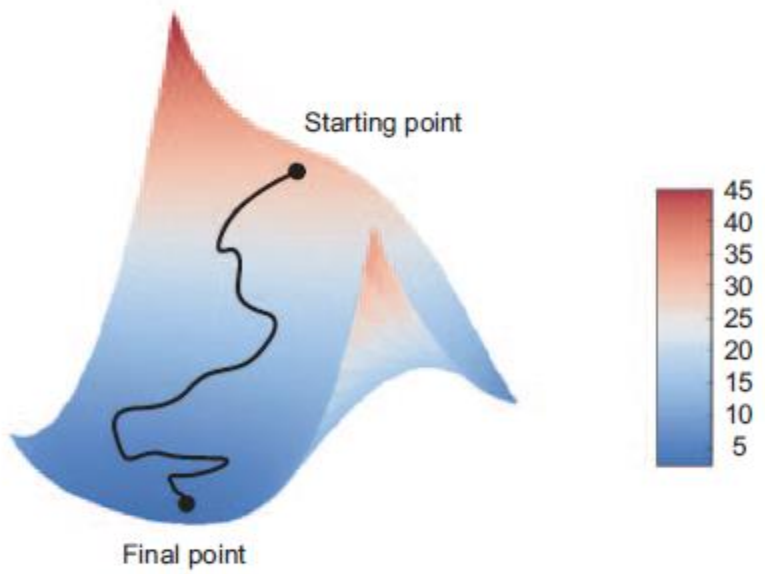


Figure 2.12 **Gradient descent down a 2D loss surface (two learnable parameters)**

2.4.3 Stochastic gradient descent

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 2.13, which shows the curve of a loss as a function of a network parameter.

2.4.3 Stochastic gradient descent

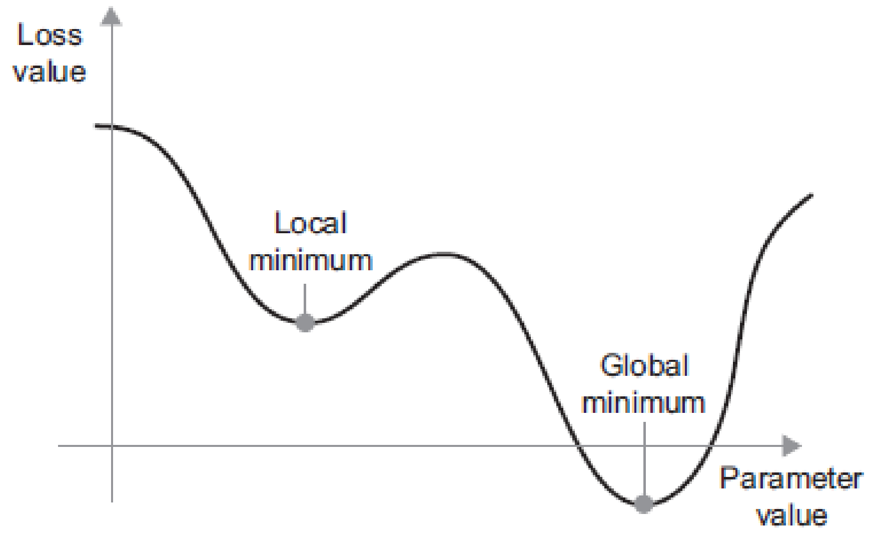


Figure 2.13 A local minimum and a global minimum

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.

2.4.3 Stochastic gradient descent

- 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 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.

2.4.3 Stochastic gradient descent

Constant momentum factor

Optimization loop

```
past_velocity = 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)
```

2.4.4 Chaining derivatives: the Backpropagation algorithm

In the previous algorithm, we casually assumed that because a function is differentiable, we can explicitly compute its derivative. In practice, a neural network function consists of many tensor operations chained together, each of which has a simple, known derivative. For instance, this is a network `f` composed of three tensor operations, `a`, `b`, and `c`, with weight matrices `W1`, `W2`, and `W3`:

```
f(W1, W2, W3) = a(W1, b(W2, c(W3)))
```

Calculus tells us that such a chain of functions can be derived using the following identity, called the *chain rule*: `f(g(x)) = f'(g(x)) * g'(x)`. Applying the chain rule to the computation of the gradient values of a neural network gives rise to an algorithm called *Backpropagation* (also sometimes called *reverse-mode differentiation*). Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value.

2.4.4 Chaining derivatives: the Backpropagation algorithm

- Nowadays, and for years to come, people will implement networks in modern frameworks that are capable of *symbolic differentiation*, such as TensorFlow.
- This means that, given a chain of operations with a known derivative, they can compute a gradient *function* for the chain (by applying the chain rule) that maps network parameter values to gradient values. When you have access to such a function, the backward pass is reduced to a call to this gradient function.
- Thanks to symbolic differentiation, you'll never have to implement the Backpropagation algorithm by hand. For this reason, we won't waste your time and your focus on deriving the exact formulation of the Backpropagation algorithm in these pages.
- All you need is a good understanding of how gradient-based optimization works.

2.5 Looking back at our first example

You’ve reached 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. 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 three 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
```

2.5 Looking back at our first example

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 network:

```
network = models.Sequential()

network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))

network.add(layers.Dense(10, activation='softmax'))
```

2.5 Looking back at our first example

Now you understand that this network 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 network persists.

This was the network-compilation step:

```
network.compile(optimizer='rmsprop’,
                loss='categorical_crossentropy’,
                metrics=['accuracy’])
```

2.5 Looking back at our first example

Now you understand that `categorical_crossentropy` is the loss function that’s used as a feedback signal for learning the weight tensors, and which the training phase will attempt to minimize. You also know that this reduction of the loss happens via minibatch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the `rmsprop` optimizer passed as the first argument.
Finally, this was the training loop:

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

2.5 Looking back at our first example

- Now you understand what happens when you call `fit`: the network 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*).
- At each iteration, the network will compute the gradients of the weights with regard to the loss on the batch, and update the weights accordingly.
- After these 5 epochs, the network will have performed 2,345 gradient updates (469 per epoch), and the loss of the network will be sufficiently low that the network 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.

Chapter summary

- *Learning* means finding a combination of model parameters 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 network parameters with respect to the loss on the batch. The network parameters are then moved a bit (the magnitude of the move is defined by the learning rate) in the opposite direction from the gradient.
- The entire learning process is made possible by the fact that neural networks are chains of differentiable tensor operations, 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.
- 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 network.
- 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.