天氣與人工智慧 II

Ch.2 The mathematical building blocks of neural networks

周哲維
david10188@gmail.com

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.

A first look at a neural network

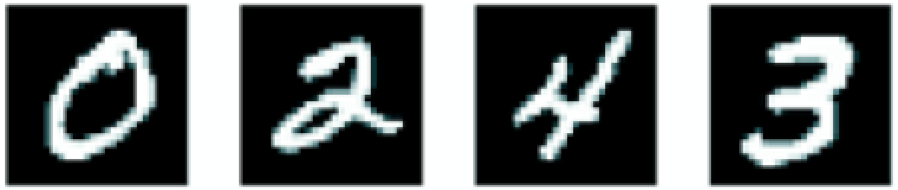


Figure 2.1 MNIST sample digits

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.

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)
```

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`.

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*.

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).

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).

A first look at a neural network

Listing 2.3 The compilation step

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

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.

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)
```

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.

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.

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?

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
```

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

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.

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
```

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.

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:

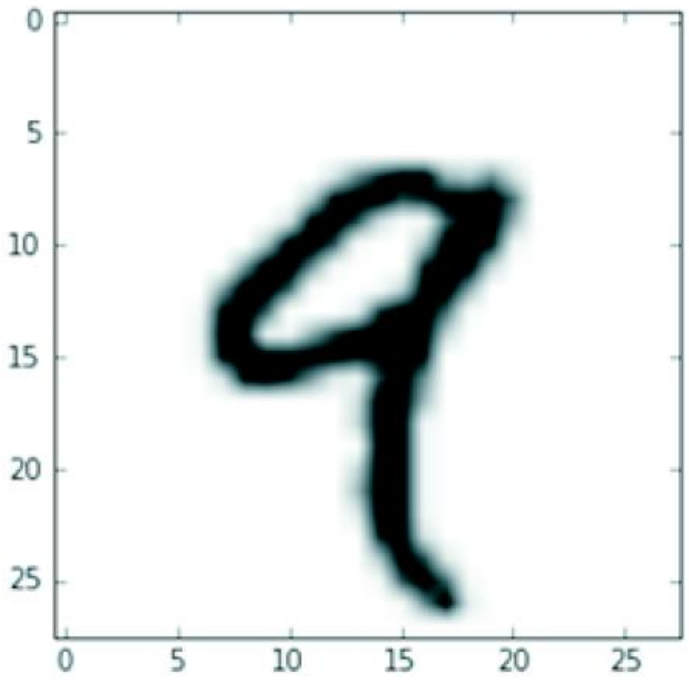
```
>>> print(train_images.ndim)
3
>>> print(train_images.shape)
(60000, 28, 28)
>>> print(train_images.dtype)
uint8
```

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.

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()
```



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)
```

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)
```

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]
```

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]
```

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`)

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*.

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.

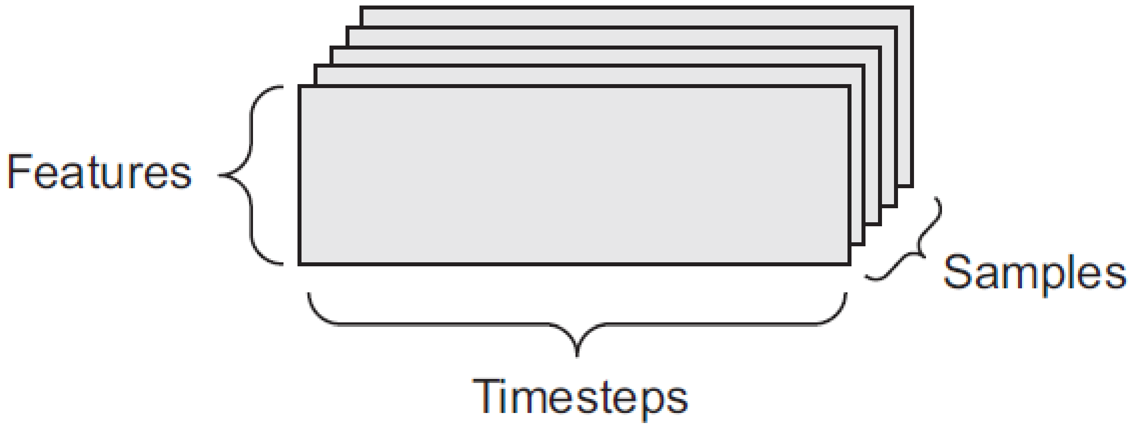
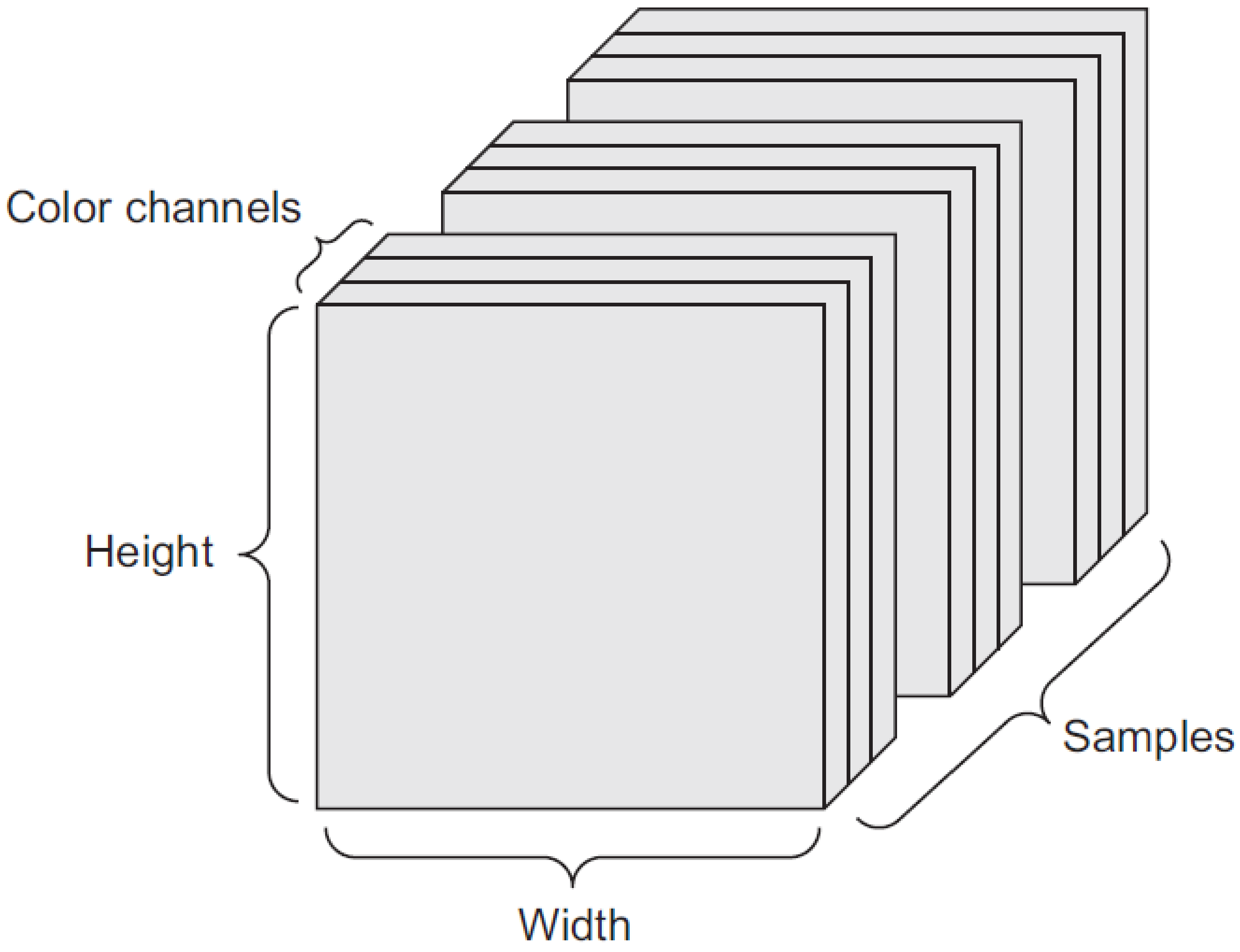


Image data

- 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)`

Image data



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)`.

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)`.

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)
```

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.

Element-wise operations

```
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 2D Numpy tensor.

Avoid overwriting the input tensor.

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.

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

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.

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.

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.

Broadcasting

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

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

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

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

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
```

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.

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.

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
```

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!

Tensor dot

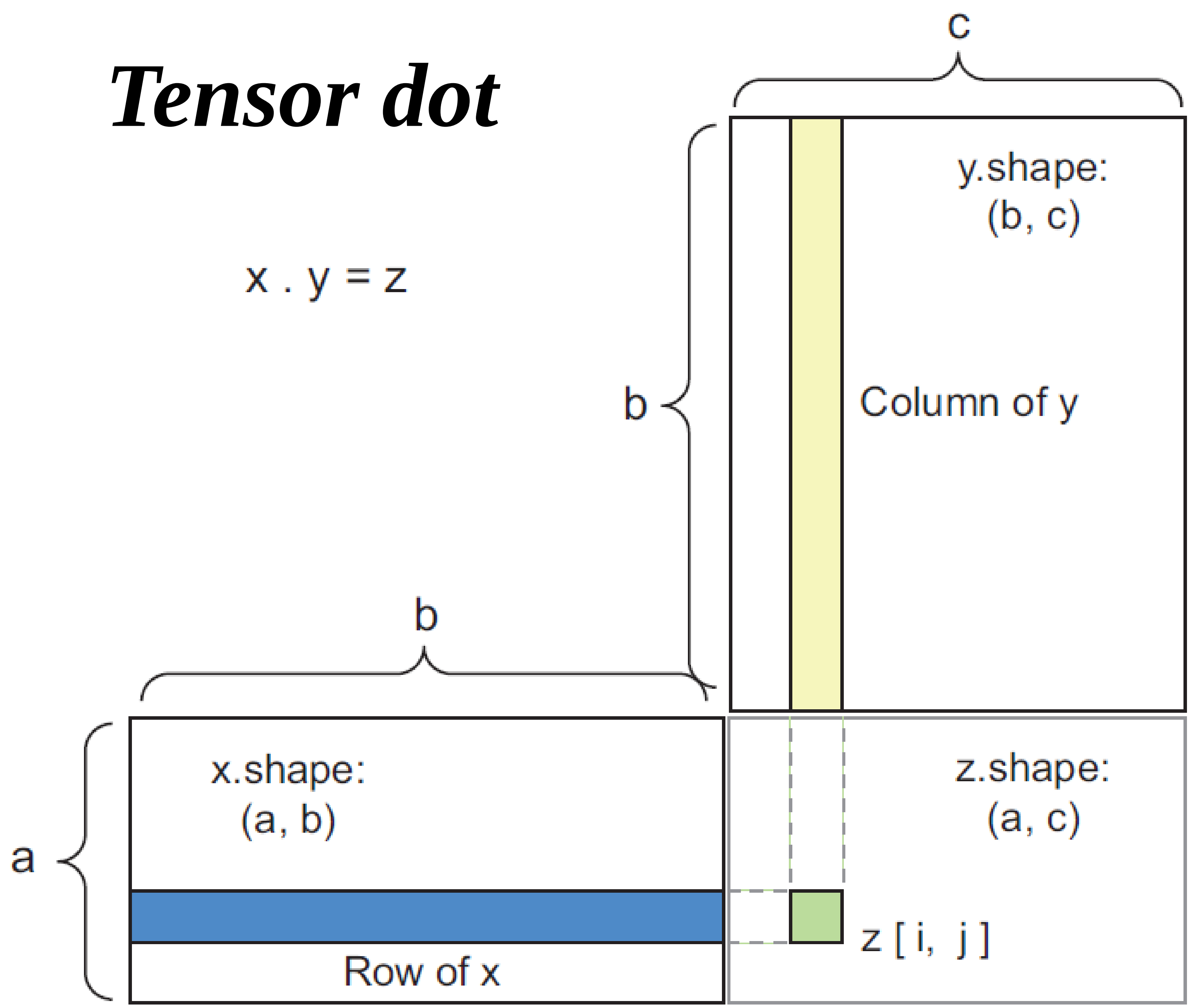


Figure 2.5 Matrix dot-product box diagram

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.

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.

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)
```

The engine of neural networks: gradient-based optimization

- 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.