Data Representation

Data representations for neural networks- ***Tensor***

- In general, all current machine-learning systems use tensors as their basic data structure.
- Tensors are fundamental to the field—so fundamental that Google's TensorFlow was named after them.
- So, what's a tensor?
- A tensor is a container for data — almost always numerical data.
- So, it's a container for numbers.
- You may be already familiar with matrices, which are 2D 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).

Data representations for neural networks- ***Scalars***

- Scalars are the 0D tensors
- A tensor that contains only one number is called a scalar (or scalar tensor, or 0-dimensional 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.

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

Data representations for neural networks- ***Vectors***

- 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, and 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

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

Data representations for neural networks- ***Matrices***

- 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.
- 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.

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

Data representations for neural networks- *3D 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.
- By packing 3D tensors in an array, you can create a 4D tensor, and so on. In deep learning, you'll generally manipulate tensors that are 0D to 4D, althc video data.

```
>>> 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 of a Tensor

- Tensor has 3 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. String tensors don't exist in NumPy (or in most other libraries), because tensors live in pre-allocated, contiguous memory segments: and strings, being variable length, would preclude the use of this implementation.

Key Attributes of a Tensor: MNIST Example

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
# we display the number of axes of the tensor train_images, the ndim attribute:
print(train_images.ndim)
```

```
3
```

```
# Here's its shape:
print(train_images.shape)
```

```
(60000, 28, 28)
```

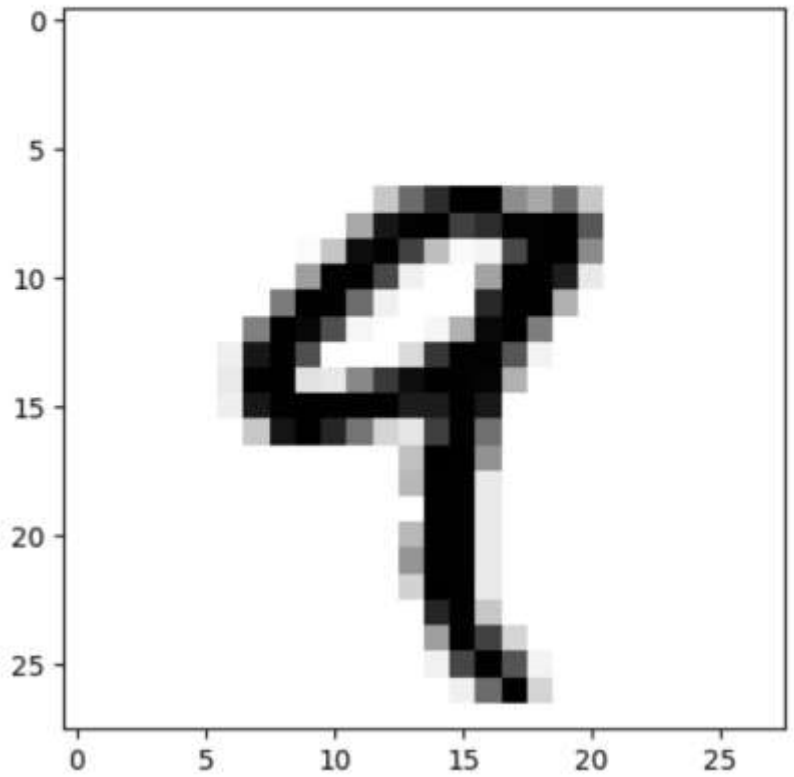
```
# And this is its data type, the dtype attribute:
print(train_images.dtype)
```

```
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 (part of the standard scientific Python suite)

Key Attributes of a Tensor: MNIST Example

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



Manipulating tensors in Numpy

- Selecting specific elements in a tensor is called tensor slicing.
- In the previous example, we selected a specific digit alongside the first axis using the syntax train_images [i].
- 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)
```

```
>>> 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

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:]
```

- 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

- 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*.

Let's take a look at two examples:

- An actuarial dataset of people, where we consider each person's age, ZIP code, and income. Each person can be characterized as a vector of 3 values, and thus an entire dataset of 100,000 people can be stored in a 2D 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 a Sequential 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 (see figure 2.3).

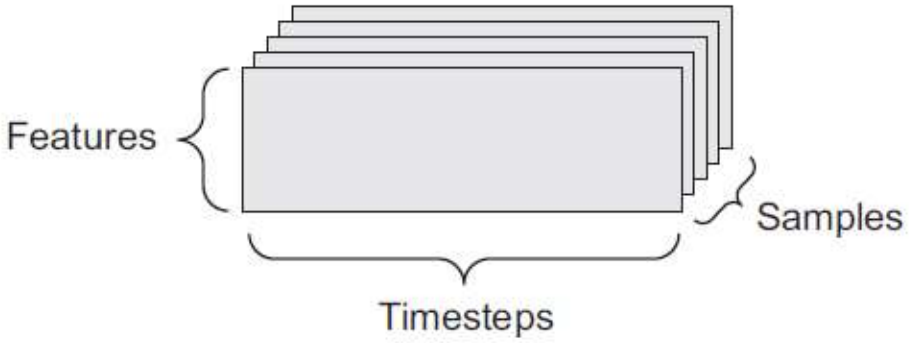


Figure 2.3 A 3D timeseries data tensor

Timeseries Data or a Sequential Data

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 2D tensor of shape `(390, 3)` (there are 390 minutes in a trading day), and 250 days' worth of data can be stored in a 3D 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 2D 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 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)` (see figure 2.4).

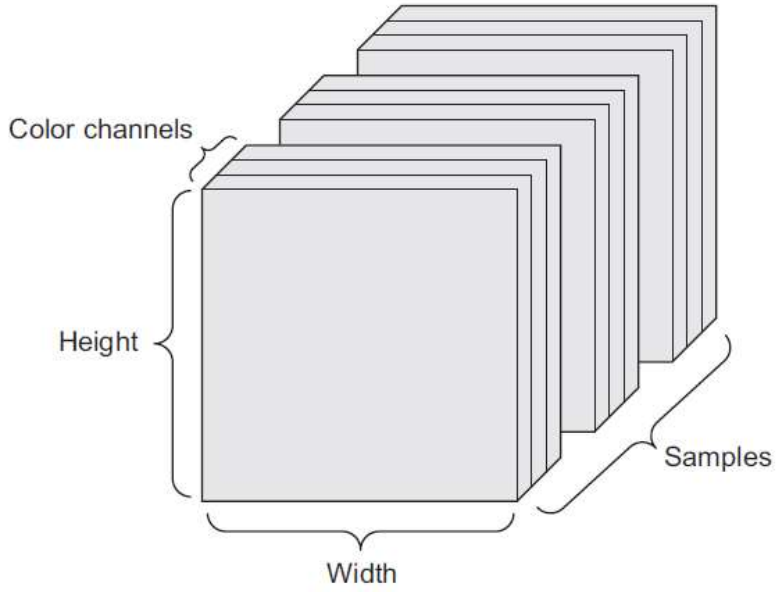


Figure 2.4 A 4D image data tensor (channels-first convention)

Video Data

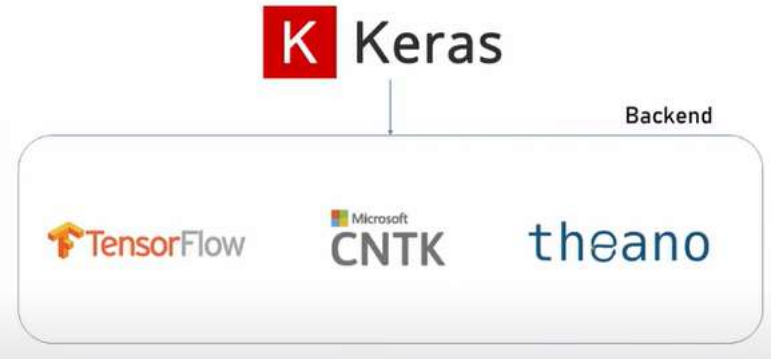
Video data is one of the few types of real-world data for which you'll need 5D tensors. A video can be understood as a sequence of frames, each frame being a color image. Because each frame can be stored in a 3D tensor (`height, width, color_depth`), a sequence of frames can be stored in a 4D tensor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

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 405 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 in the MPEG format).

PyTorch versus TensorFlow versus Keras

- PyTorch and TensorFlow are two popular Deep Learning Frameworks
- There is 3rd framework called CNTK by Microsoft but it is less popular
- PyTorch developed by FaceBook and TensorFlow by Google
- Keras is wrapper for TensorFlow CN

PyTorch versus TensorFlow versus Keras




```
import keras
keras.backend.backend()  'tensorflow'

from keras.models import Sequential
from keras.layers import Flatten, Dense, Activation

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

```
from tensorflow import keras

model = keras.Sequential([
    keras.layers.Dense(10, input_shape=(784,), activation='sigmoid')
])
```