Data representations for neural networks

13/12/2021

```
library(keras)
```

Data representations for neural networks

In general, all current machine-learning systems use multidimensional arrays, also called 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?

Tensors are a generalization of vectors and matrices to an arbitrary number of dimensions (note that in the context of tensors, "dimension" is often called "axis"). Within R, vectors are used to create and manipulate 1D tensors and matrices are used for 2D tensors. For higher level dimensions array objects (which support any number of dimensions) are used.

Scalars (0D tensors)

A tensor that contains only one number is called a scalar (or scalar tensor, or 0-dimensional tensor, or 0D tensor). While R does not have a data type to represent scalars (all numeric objects are vectors, matrices, or arrays), an R vector that is always length 1 is conceptually similar to a scalar.

Vectors (1D tensors)

A one dimensional array of numbers is called a vector, or 1D tensor. A 1D tensor is said to have exactly one axis. We can convert the R vector to an array object in order to inspect its dimensions:

```
x <- c(12, 3, 6, 14, 10)
str(x)
```

```
##  num [1:5] 12 3 6 14 10
```

```
dim(as.array(x))
```

```
## [1] 5
```

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.

Matrices (2D tensors)

A two dimensional array of numbers is a matrix, or 2D tensor. A matrix has two axes (often referred to rows and columns).

```
x <- array(rep(0, 6), dim = c(2,3))
str(x)
```

```
##  num [1:2, 1:3] 0 0 0 0 0 0
```

```
dim(x)
```

```
## [1] 2 3
```

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:

```
x <- array(rep(0, 2*3*2), dim = c(2,3,2))
str(x)
```

```
##  num [1:2, 1:3, 1:2] 0 0 0 0 0 0 0 0 0 0 ...
```

```
dim(x)
```

```
## [1] 2 3 2
```

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 form 0D to 4D, although you may go up to 5D if you process video data.

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.
- Shape —This is an integer vector that describes how many dimensions the tensor has along each axis. For instance, the previous matrix example has shape (2, 3), and the 3D tensor example has shape (2, 3, 2). A vector has a shape with a single element, such as (5). You can access the dimensions of any array using the dim() function.
- Data type —This is the type of the data contained in the tensor; for instance, a tensor's type could be integer or double. On rare cases, you may see a character tensor. However, since tensors live in pre-allocated contiguous memory segments, and strings, being variable-length, would preclude the use of this implementation, they are more rarely used.

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

```
mnist <- dataset_mnist()
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

Next we display the number of axes of the tensor train_images:

```
length(dim(train_images))
```

```
## [1] 3
```

Here's its shape:

```
dim(train_images)
```

```
## [1] 60000    28    28
```

And this is its data type:

```
typeof(train_images)
```

```
## [1] "integer"
```

So what we have here is a 3D tensor of 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 plot the 5th digit in this 3D tensor:

```
digit <- train_images[5,,]
plot(as.raster(digit, max = 255))
```

Manipulating tensors in R

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 take a look at the tensor slicing operations that you can do on R arrays. The following chunk selects from digits #10 to #99 and puts them in an array of shape (90, 28, 28):

```
my_slice <- train_images[10:99,,]
dim(my_slice)
```

```
## [1] 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.

```
my_slice <- train_images[10:99,1:28,1:28]
dim(my_slice)
```

```
## [1] 90 28 28
```

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'd do this:

```
my_slice <- train_images[, 15:28, 15:28]
```

The notion of data batches

In general, the first axis 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[1:128,,]
batch <- train_images[129:256,,]
```

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)

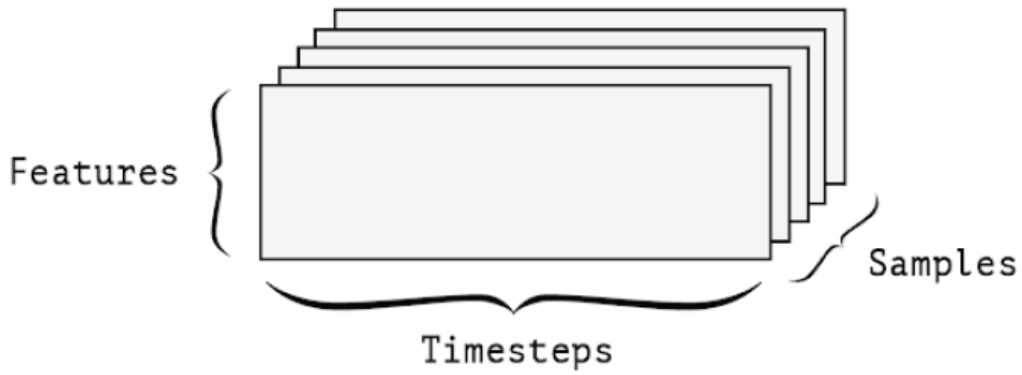
Vector data

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. 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 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

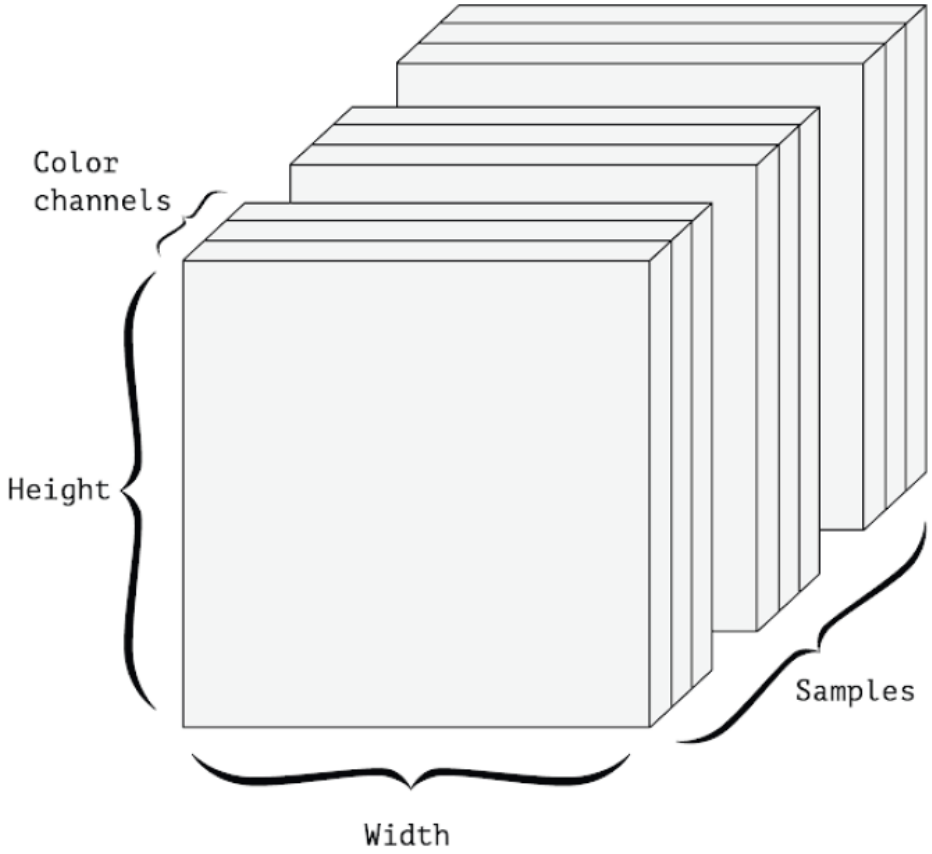


The time axis is always the second axis, by convention. Let's have a look at a few examples:

- 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 140 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 (140, 128), and a dataset of 1 million tweets can be stored in a tensor of shape (1000000, 140, 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)



There are two conventions for shapes of images tensors: the channels-last convention (used by TensorFlow) and the channels-first convention (used by Theano). The TensorFlow machine-learning framework, from Google, places the color-depth axis at the end, as you just saw: (samples, height, width, color_depth). Meanwhile, Theano places the color depth axis right after the batch axis: (samples, color_depth, height, width). With the Theano convention, the previous examples would become (128, 1, 256, 256) and (128, 3,

256, 256). The Keras framework provides support for both formats.

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 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, 256 × 144 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, 256, 144, 3). That's a total of 106,168,320 values! If the data type of the tensor is double, then each value is stored in 64 bits, so the tensor would represent 810 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).

Tensor reshaping

A tensor operation that's essential to understand is tensor reshaping We used it when we preprocessed the digits data before feeding them into our network:

```
train_images <- array_reshape(train_images, c(60000, 28 * 28))
```

Note that we use the array_reshape() function rather than the dim<-() function to reshape the array. This is so that the data is re-interpreted using row-major semantics (as opposed to R's default column-major semantics), which is in turn compatible with the way that the numerical libraries called by Keras (e.g. NumPy, TensorFlow, etc.) interpret array dimensions. You should always use the array_reshape() function when reshaping R arrays that will be passed to Keras. 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 <- matrix(c(0, 1,
              2, 3,
              4, 5),
nrow = 3, ncol = 2, byrow = TRUE)
 x
```

```
##      [,1] [,2]
## [1,]    0    1
## [2,]    2    3
## [3,]    4    5
```

```
x <- array_reshape(x, dim = c(6, 1))
x
```

```
##      [,1]
## [1,]    0
## [2,]    1
## [3,]    2
## [4,]    3
## [5,]    4
## [6,]    5
```

```
x <- array_reshape(x, dim = c(2, 3))
x
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    2
## [2,]    3    4    5
```