Foundations for Machine Learning

L. Y. Stefanus

TU Dresden, June-July 2019

Slide 04p

Real-World Data Representation Using Tensors

Reference

- Eli Stevens and Luca Antiga. Deep Learning with PyTorch. Manning Publications, 2019/2020.
- Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning. MIT Press, 2016.

Tensor as Building-blocks

- We have learned that tensors are the building blocks for data in PyTorch.
- Neural networks take tensors as input and produce tensors as outputs. In fact, all operations within a neural network and during optimization are operations between tensors, and all parameters (e.g. weights) in a neural network are tensors.
- How do we take a piece of data, a video, or some text, and represent it with a tensor that is appropriate for training a deep learning model?

Tabular Data

- The simplest form of data: in a spreadsheet, in a CSV (comma-separated values) file, or in database.
- Whatever the medium, it's a table containing one row per sample (or record), where a column contains one piece of information of a sample.
- At first we assume there's no meaning in the order in which samples appear in the table: such table is a collection of independent samples, unlike a time-series, for instance, in which samples are related by a time dimension.

Tabular Data

- Columns may contain numerical values, like temperatures at specific locations, or labels, like a string expressing an attribute of the sample, like “green". Therefore, tabular data is typically not homogeneous: different columns don’t have the same type.
- PyTorch tensors, on the other hand, are homogeneous.
- Information in PyTorch is typically encoded as a number. This numeric encoding is deliberate, since neural networks are mathematical entities that take real numbers as inputs and produce real numbers as output through successive application of matrix multiplications and non-linear functions.

Tabular Data

- Our first job, as deep learning practitioners, is therefore to encode heterogenous, real-world data into a tensor of floating point numbers, ready for consumption by a neural network.
- There are a large number of tabular datasets freely available on the Internet, see for instance: github.com/caesar0301/awesome-public-datasets
- We will use the Wine Quality dataset, which can be downloaded from here: archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv.

Tabular Data

- The file contains a comma-separated collection of values organized in 12 columns preceded by a header line containing the column names. The first 11 columns contain values of chemical variables, while the last column contains the sensory quality score from 0 (very bad) to 10 (excellent).
- These are the column names in the order they appear in the dataset: fixed acidity, volatile acidity, citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates, alcohol, quality.
- A possible machine learning task on this dataset is predicting the quality score from chemical characterization alone.

Tabular Data

- As we can see in Figure-4.1, we're expecting to see quality increase as sulfur decreases.

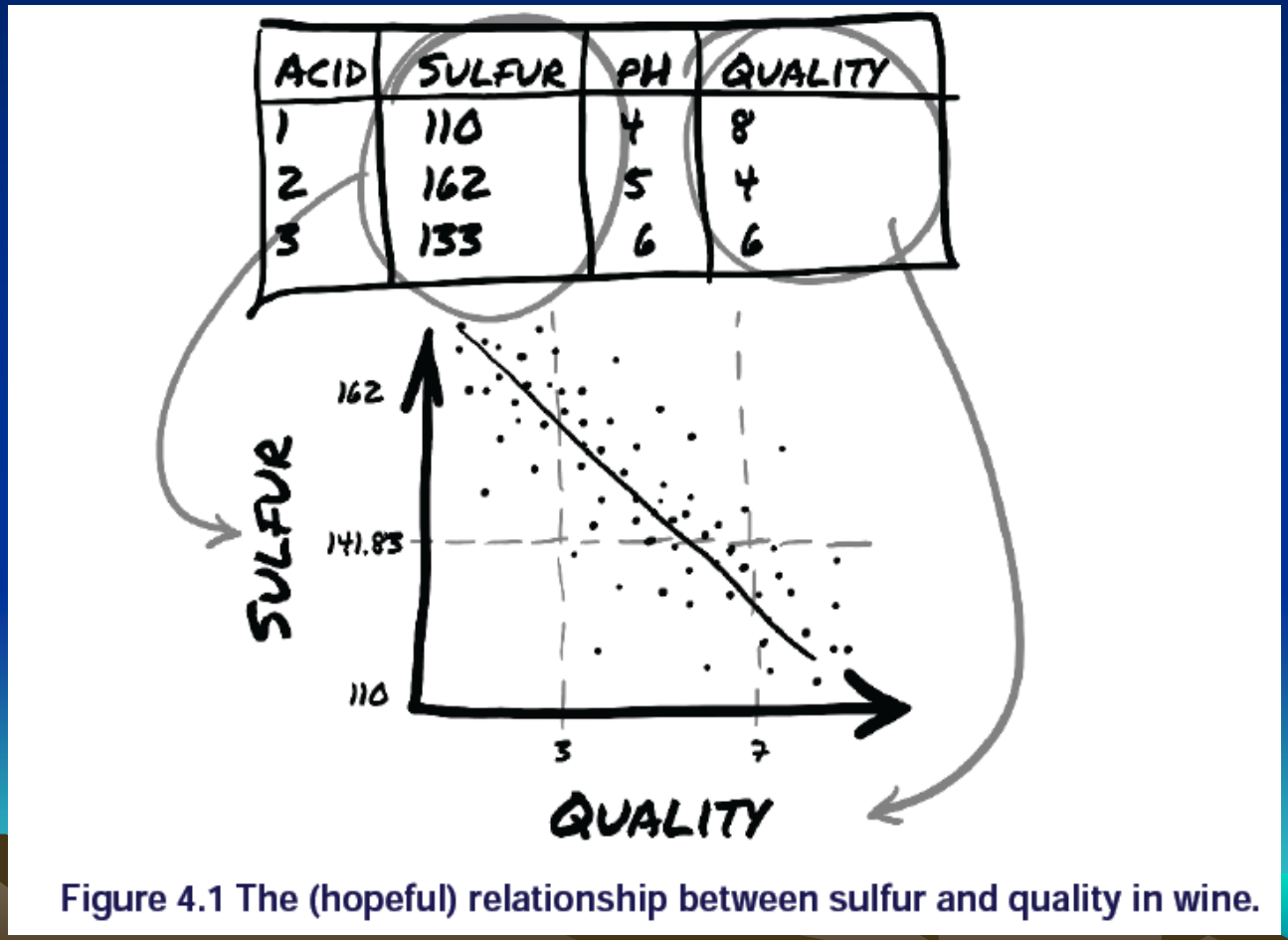

Figure 4.1 The (hopeful) relationship between sulfur and quality in wine.

Tabular Data

- Let's see how we can load the data using Python and then turn it into a PyTorch tensor. Python offers several options for quickly loading a CSV file: `csv` module, NumPy, Pandas.
- Since PyTorch has excellent NumPy interoperability, we'll go with that. Let's load our file and turn the resulting NumPy array into a PyTorch tensor.

Tabular Data

```
# In[2]:
import csv
Import numpy as np
wine_path = "C:/Kuliah/machineLearning2019/data/winequality.csv"
wineq_numpy = np.loadtxt(wine_path, dtype=np.float32,
    delimiter=";", skiprows=1)
wineq_numpy
# Out[2]:
array([[ 7.  , 0.27, 0.36, ..., 0.45, 8.8 , 6. ],
[ 6.3 , 0.3 , 0.34, ..., 0.49, 9.5 , 6. ],
[ 8.1 , 0.28, 0.4 , ..., 0.44, 10.1 , 6. ],
...,
[ 6.5 , 0.24, 0.19, ..., 0.46, 9.4 , 6. ],
[ 5.5 , 0.29, 0.3 , ..., 0.38, 12.8 , 7. ],
[ 6.  , 0.21, 0.38, ..., 0.32, 11.8 , 6. ]], dtype=float32)
```

- For the function loadtxt from Numpy, we just prescribed what the type of the 2D array should be (32-bit floating point), the delimiter used to separate values in each row and the fact that the first line should not be read since it contains the column names.

Tabular Data

```
# In[3]:
col_list = next(csv.reader(open(wine_path), delimiter=';'))

winеq_numpy.shape, col_list

# Out[3]:
((4898, 12),
 ['fixed acidity',
  'volatile acidity',
  'citric acid',
  'residual sugar',
  'chlorides',
  'free sulfur dioxide',
  'total sulfur dioxide',
  'density',
  'pH',
  'sulphates',
  'alcohol',
  'quality'])

# In[4]:
wineq = torch.from_numpy(wineq_numpy)

wineq.shape, wineq.type()

# Out[4]:
(torch.Size([4898, 12]), 'torch.FloatTensor')
```

At this point we have a torch.FloatTensor containing all columns of the table.

Tabular Data

- We will remove the quality score from the tensor of input data and keep it in a separate tensor, so that we can use the score as the ground truth.

```
# In[5]:
data = wineq[:, :-1]    ❶
data, data.shape

# Out[5]:
(tensor([[ 7.00,  0.27,  ...,  0.45,  8.80],
         [ 6.30,  0.30,  ...,  0.49,  9.50],
         ...,
         [ 5.50,  0.29,  ...,  0.38, 12.80],
         [ 6.00,  0.21,  ...,  0.32, 11.80]]), torch.Size([4898, 11]))

# In[6]:
target = wineq[:, -1]    ❷
target, target.shape

# Out[6]:
(tensor([6., 6.,  ..., 7., 6.]), torch.Size([4898]))
```

1. select all rows,
all columns except the last
2. select all rows,
the last column

Tabular Data

- If we want to transform the target tensor into a tensor of labels, we have two options, depending on the strategy or what we use the categorical data for. One is simply to treat labels as an integer vector of scores:

```
# In[7]:
target = wineq[:, -1].long()
target

# Out[7]:
tensor([6, 6,  ..., 7, 6])
```

- If targets were string labels, like *colors*, assigning an integer number to each string would allow for the same approach.

Tabular Data

One-hot Encoding

- The other approach is to build a *one-hot* encoding of the quality scores, that is, encode each of the 10 scores in a vector of 10 elements, with all elements set to zero except one, at a different index for each score. This way a score of 1 could be mapped onto the vector (1,0,0,0,0,0,0,0,0,0), a score of 5 onto (0,0,0,0,1,0,0,0,0,0) and so on.

Tabular Data

- There's an important difference between the two approaches.
- Keeping wine quality scores in an integer vector of scores induces an ordering on the scores - which might be appropriate in this case, since a score of 1 is lower than a score of 4. It also induces some sort of distance between scores, i.e. the distance between 1 and 3 is the same as the distance between 2 and 4. If this holds for our quantity, then fine. If, on the other hand, scores were purely qualitative, like colors, one-hot encoding will be a much better fit, as there's no implied ordering or distance.

Tabular Data

- One-hot encoding will also be appropriate for quantitative scores when fractional values in-between integer scores, like 2.4, make no sense for the application.
- We can achieve one-hot encoding using the scatter_ method, which fills the tensor with values from a source tensor along the indices provided as arguments.

```
# In[8]:
target_onehot = torch.zeros(target.shape[0], 10)
target_onehot.scatter_(1, target.unsqueeze(1), 1.0)
target_onehot[0]
# Out[8]:
tensor([0., 0., 0., 0., 0., 0., 1., 0., 0., 0.])
```

Tabular Data

- The arguments for scatter_ are:
  1. The dimension along which the following two arguments are specified
  2. A column tensor indicating the indices of the elements to scatter
  3. A tensor containing the elements to scatter or a single scalar to scatter (1.0 in this case)
- In other words, the above invocation reads: for each row, take the index of the target label (which coincides with the score in our case) and use it as the column index to set the value 1.0.
- The end-result is a tensor encoding categorical information.

Tabular Data

- The second argument of scatter_, the index tensor, is required to have the same number of dimensions as the tensor we scatter into. Since target_onehot has two dimensions (4898x10), we needed to add an extra dummy dimension to target using the method unsqueeze.

```
# In[9]:
target_unsqueezed = target.unsqueeze(1)
target_unsqueezed

# Out[9]:
tensor([[6],
        [6],
        ...,
        [7],
        [6]])
```

Tabular Data

- The call to unsqueeze added a *singleton* dimension, from a 1D tensor of 4898 elements to a 2D tensor of size (4898x1), without changing its contents - there are no extra elements added, we just use an extra index to access the elements. For example, we access the first element of target as target[0] and the first element of its unsqueezed counterpart as target_unsqueezed[0,0].
- If we want to use the score as a categorical input to a neural network in PyTorch, we would have to transform it to a *one-hot* encoded tensor.

Images

- The introduction of convolutional neural networks revolutionized computer vision. In order to work in computer vision, we need to be able to load images from common image formats, and then transform the data into a tensor representation that has the various parts of the image arranged in the way that PyTorch expects.
- An image is represented as a collection of scalars arranged in a regular grid, having a height and a width (in pixels). One might have a single scalar per grid point (the pixel), which would be represented as a grayscale image, or multiple scalars per grid point, which would typically be representing different colors, or different features like depth from a depth camera.

Images

- Scalars representing values at individual pixels are often encoded using 8-bit integers, for instance in consumer cameras. In medical, scientific or industrial applications it is not infrequent to find pixels with higher numerical precision, like 12-bit or 16-bit.
- There are several ways of encoding colors into numbers. The most common is RGB, where a color is defined by three numbers representing the intensity of red, green and blue. We can think of a color channel as a grayscale intensity map of only the color in question, similar to what you'd see if you looked at the scene in question using a pair of pure red sunglasses.

Images

- Images come in several different file formats, but there are many ways of loading images in Python. Let's start loading a PNG image using the imageio module, which can handle different data types with a uniform API.

```
# In[2]:
import imageio
img_arr = imageio.imread('C:/Kuliah/ml/data/bobby.jpg')
img_arr.shape
# Out[2]:
(720, 1280, 3)
```

Images

- At this point, img_arr is a NumPy array object with three dimensions: two spatial dimensions, width and height, and a third dimension corresponding to channels (red, green and blue).
- Any library that outputs a NumPy array will do in order to obtain a PyTorch tensor. The only thing to watch out for is the layout of dimensions. PyTorch modules dealing with image data require tensors to be laid out as C x H x W (channels, height, width).
- We can use the transpose function to get to an appropriate layout. Given an input tensor with layout W x H x C as obtained above, we get to a proper layout by swapping the first and last dimensions:

```
# In[3]:
img = torch.from_numpy(img_arr)
out = torch.transpose(img, 0, 2)
```

- Note that the two tensors use the same storage.

Images

- So far we have described a single image. In order to create a dataset of multiple images to use as an input for our neural networks, we store the images in a batch along the first dimension to obtain a N x C x H x W tensor.
- As a more efficient alternative to using stack to build up the tensor, we can pre-allocate a tensor of appropriate size and fill it with images loaded from a directory, like so:

```
# In[4]:
batch_size = 100
batch = torch.zeros(100, 3, 256, 256, dtype=torch.uint8)
```

```
# In[5]:
import os

data_dir = '../data/p1ch4/image-cats/'
filenames = [name for name in os.listdir(data_dir) if os.path.splitext(name) == '.png']
for i, filename in enumerate(filenames):
    img_arr = imageio.imread(filename)
    batch[i] = torch.transpose(torch.from_numpy(img_arr), 0, 2)
```

Images

- Our batch or dataset consists of 100 RGB images of 256 pixels in height and 256 pixels in width. Notice the type of the tensor: we're expecting each color to be represented as a 8-bit integer, as in most photographic formats from standard consumer cameras.
- Neural networks exhibit the best training performance when input data ranges roughly from 0 to 1 (or from -1 to 1) in floating-point values.
- So a typical thing we'll want to do is cast a tensor to floating point and normalize the values of the pixels. One possibility is to just divide the values of pixels by 255 (the maximum representable number in 8-bit unsigned):

```
# In[6]:
batch = batch.float()
batch /= 255.0
```