Real-world data representation using tensors

This chapter covers:

- Representing real-world data as PyTorch tensors
- Working with a range of data types
- Loading data from a file
- Converting data to tensors
- Shaping tensors so they can be used as inputs for neural network models

Scalars representing pixels are usually 8-bit integers

- in industry, higher precision such as 12-bit and 16-bit are also possible

Importing an image using `imageio` module

```
# In[2]:
import imageio
img_arr = imageio.imread('../data/p1ch4/image-dog/bobby.jpg')
img_arr.shape

# Out[2]:
(720, 1280, 3)
```

Note: `imageio` will be used throughout the chapter, because it handles different data types with uniform API

- for many purposes, TOrchVision is a great default choice to deal with image/video data
  - `dataset = datasets.ImageFolder('path', transform=transform)`
- imageio is just a lighter version

Watch our for the layout of the images

- PyTorch modules dealing with img data requires tensors to be laid out as C X H X W
- (Tensorflow supports multiple layouts)
- `imageio` returns as H X W X C

```
# In[3]:
img = torch.from_numpy(img_arr)
out = img.permute(2, 0, 1)
```

`out` uses the same underlying storage as `img`

- Changing a pixel in `img` will lead to a change in `out`

Slightly more efficient alternative to using `stack` to build up the input tensor

- Can preallocate a tensor of appropriate size and fill it with images loaded from a dir

```
# In[4]:
batch_size = 3
batch = torch.zeros(batch_size, 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
)[-1] == '.png']

for i, filename in enumerate(filenames):
    img_arr = imageio.imread(os.path.join(data_dir, filename))
    img_t = torch.from_numpy(img_arr)
    img_t = img_t.permute(2, 0, 1)
    img_t = img_t[:3] # Keep only first 3 channels. Sometimes there is an a
dditional channel (transparency)
    batch[i] = img_t
```

Above example assumes each color will be represented in 8-bit integers

Normalizing the data

- Neural networks usually work with floating-point tensors as their input
- Neural networks **exhibit best training performance when the input data ranges from rougly 0 to 1, or from -1 to 1**

One possibility is just to divide the values of pixels by 255

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

Another possibility is to compute the mean and std of the input data and scale it

- 0 mean, unit std across each channel

```
# In[7]:
n_channels = batch.shape[1]
for c in range(n_channels):
mean = torch.mean(batch[:, c])
std = torch.std(batch[:, c])
batch[:, c] = (batch[:, c] - mean) / std
```

- good practice to compute the mean and standard deviation on all the training data in advance and then subtract nd divide by these fixed, precomputed quantities.

3D images have an additional channel for depth : `N x C x D x H x W`

- Can use specialized functions to load specialized format
  - ex) `imageio.volread`

Representing Tabular Data

Our first job as deep learning practitioners is to encode heterogeneous, real-world data into a tensor of floating-point numbers, ready for consumption by a neural network

Popular options for loading CSV files:

- The csv module that ships with Python
- NumPy
- Pandas (most time- and memory- efficient)

```
# In[2]:
import csv
wine_path = "../data/p1ch4/tabular-wine/winequality-white.csv"
wineq_numpy = np.loadtxt(wine_path, dtype=np.float32, delimiter=";",skiprow
s=1)
# first row contains column names

# check that all data has been read
col_list = next(csv.reader(open(wine_path), delimiter=';'))
wineq_numpy.shape, col_list

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

#convert NumPy array to PyTorch tensor
# In[4]:
wineq = torch.from_numpy(wineq_numpy)
wineq.shape, wineq.dtype
# Out[4]:
(torch.Size([4898, 12]), torch.float32)
```

Continuous, ordinal, categorial values

- Continuous:
  - strictly ordered, difference between values have strict meaning
  - distance, etc
- Ordinal
  - Strictly ordered, difference has no strict meaning
  - sizes "small, medium, large"
- Categorical
  - No strict order, difference has no strict meaning

- assigning water to 1, coffee to 2, soda to 3...
- best handled by one-hot encoding

> Further explanations and code are omitted (focusing on computer vision), Refer to original text and code repo.

```
"""
Other mentioned pytorch functionalities
"""

# scatter_ method for one-hot encoding
target_onehot.scatter_(1, target.unsqueeze(1), 1.0)
# 1 : The dimension along which the following two arguments are specified
# target.unsqueeze(1) : Column tensor indicating indices of elements to sca
tter (need to be 0-indexed)
#                       Added a singleton dimension by using unsqueeze(1)
# 1.0 : Tensor containing the elements to scatter or a single scalar to sca
tter (usually set to 1 for one-hot)

# Comparison functions in tensors
bad_indexes = target <= 3
bad_indexes.shape, bad_indexes.dtype, bad_indexes.sum()

#Boolean operations for Boolean NumPy arrays and PyTorch Tensors
mid_data = data[(target > 3) & (target < 7)]

# lt (less than) operations for thresholding
total_sulfur_threshold = 141.83
total_sulfur_data = data[:,6]
predicted_indexes = torch.lt(total_sulfur_data, total_sulfur_threshold)

# .item() method to just get the value of the tensor
n_matches = torch.sum(actual_indexes & predicted_indexes).item()
n_predicted = torch.sum(predicted_indexes).item()
n_actual = torch.sum(actual_indexes).item()
```

Working with Time series

> Omitted

```
# Other mentioned pytorch functionalities

# concatenation of tensors
torch.cat((bikes[:24], weather_onehot), 1)[:1] # [:1] to just show one entr
y
# concatenated along the column dimension (dimension 1)
# Therefore the two tensors are just stacked (appended to the original data
set)
# For cat to succeed, required that the tensors have the same size along th
e other dimensions(batch, row...)

# More specific example, appending to a tensor of dimension B x C x L
# In[9]:
daily_weather_onehot = torch.zeros(daily_bikes.shape[0], 4, daily_bikes.sha
pe[2]) #Match B and L, newly define C
daily_weather_onehot.shape
# Out[9]:
torch.Size([730, 4, 24])

# In[10]:
daily_weather_onehot.scatter_(
    1, daily_bikes[:,9,:].long().unsqueeze(1) - 1, 1.0)
daily_weather_onehot.shape
# Out[10]:
torch.Size([730, 4, 24])

# In[11]:
daily_bikes = torch.cat((daily_bikes, daily_weather_onehot), dim=1)

# Or just treat the ordinal variables as a continuous relationship:
# In[12]:
daily_bikes[:, 9, :] = (daily_bikes[:, 9, :] - 1.0) / 3.0 # normalized from
0.0 ~ 1.0
```

Representing Text

Omitted

```
# Other mentioned pytorch functionalities

# Representing words as one-hot encoded vectors
word_list = sorted(set(clean_words(text)))
word2index_dict = {word: i for (i, word) in enumerate(word_list)}
len(word2index_dict), word2index_dict['impossible']
# Out[7]:
(7261, 3394)

# Create an empty vector and assign one-hot encoded values of the word in t
he sentence

# In[8]:
word_t = torch.zeros(len(words_in_line), len(word2index_dict))
for i, word in enumerate(words_in_line):
word_index = word2index_dict[word]
word_t[i][word_index] = 1
print('{:2} {:4} {}'.format(i, word_index, word))
print(word_t.shape)
# Out[8]:
0 3394 impossible
1 4305 mr
2 813 bennet
3 3394 impossible
4 7078 when
5 3315 i
6 415 am
7 4436 not
8 239 acquainted
9 7148 with
10 3215 him
torch.Size([11, 7261])
```