Tensor_Data_Processing

January 16, 2024

1 Data Manipulation

- The basic data structure used in deep learning is the n-dimensional array, which is also called a *tensor*.
  - A rank 0 tensor corresponds to a *number*.
  - A rank 1 tensor corresponds to a *vector*.
  - A rank 2 tensor corresponds to a *matrix*.
  - Tensors of higher rank do not have special names.
- The class `Tensor` in PyTorch is similar to the class `ndarray` in Numpy. However, it also enables GPU acceleration and automatic differentiation.
- Together, these properties make the `Tensor` class very useful for deep learning.

[1]:
```
# First, we import `torch` (the library is called PyTorch, but it is imported
 ↪as `torch`)
import torch
```

[2]:
```
print(torch.__version__)
```

```
2.1.0+cu121
```

2 Vector

- We will denote vectors by boldface lower-case letters (such as $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).
- Operations between matrices and vectors will behave as if the vector were a column matrix:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where $x_1, \ldots, x_n$ are elements of the vector.

[3]:
```
# A vector with 4 integers in the range [0,3]
# Unless otherwise specified, a new tensor is stored in main memory, enabling
 ↪CPU-based computation
x = torch.arange(4)
print(type(x))
print(x)
```

```
<class 'torch.Tensor'>
tensor([0, 1, 2, 3])
```

[4]:
```python
# Acessing the i-th element: x[i]. Indices start at zero.
print(x[3])
```

```
tensor(3)
```

[5]:
```python
# Vector shape (dimensionality)
print(len(x))
print(x.size())
print(x.shape)
print(type(x.size()))
```

```
4
torch.Size([4])
torch.Size([4])
<class 'torch.Size'>
```

3 Matrices

- We will denote matrices by boldface capital letters (such as $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$).
- We will let $\mathbf{A} \in \mathbb{R}^{m \times n}$ denote that matrix $\mathbf{A}$ is composed of m rows and n columns of real numbers.
- We can represent a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ by a table:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},$$

where the element in the i^{th} row and j^{th} column is given by a_{ij}.

- For any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the *shape* of $\mathbf{A}$ is (m, n).
- A matrix is called a *square matrix* if the number of rows is the same as the number of columns.

[6]:
```python
# Reshape function: changes the shape of a tensor without changing the number␣
↪of elements or their values
A = torch.arange(20)
print(A)
B = A.reshape(5, 4)
print(B)
print(B[2, 3])
print(B[2][3])
```

```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
        18, 19])
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]])
tensor(11)
tensor(11)
```

[7]:
```
# Reshaping convention
A = torch.arange(20)
print(A)

# From vector to matrix
c = 0
B = torch.zeros((5, 4), dtype=int)
for i in range(5):
    for j in range(4):
        B[i, j] = A[c]
        c += 1
print(B)

# From matrix to vector
c = 0
C = torch.zeros(20, dtype=int)
for i in range(5):
    for j in range(4):
        C[c] = B[i, j]
        c += 1
print(C)
```

```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
        18, 19])
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]])
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
        18, 19])
```

[8]:
```
# Matrix shape
print(len(B))
print(B.size())
print(B.shape)
```

```
5
torch.Size([5, 4])
torch.Size([5, 4])
```

[9]:
```
# Use -1 for a dimension that can be automatically inferred
A1 = torch.arange(20).reshape(5, -1);
A2 = torch.arange(20).reshape(-1, 4);
print(A1 == A2)
```

```
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```

[10]:
```
# Transpose
B1 = B.T
print(B1)
B2 = B1.transpose(1, 0)
print(B2)
```

```
tensor([[ 0,  4,  8, 12, 16],
        [ 1,  5,  9, 13, 17],
        [ 2,  6, 10, 14, 18],
        [ 3,  7, 11, 15, 19]])
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]])
```

4 Tensors

[11]:
```
# A rank 3 tensor
X = torch.arange(24).reshape(2, 3, -1)
print(X)
```

```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

5 Commonly-used Tensor Constructors

[12]:
```
print(torch.ones((2, 3, 4))) # Initialize with ones
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
```

```
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

[13]:
```python
print(torch.zeros(2, 3)) # Initialize with zeros
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

[14]:
```python
print(torch.randn(3, 4)) # Initialize with samples from a Gaussian distribution
 ↪with mean 0 and standard deviation 1
```

```
tensor([[-0.5730,  0.0458, -1.0057, -0.2085],
        [ 0.3169, -0.1660, -0.0406,  0.0657],
        [ 0.0972, -0.1748, -1.0358, -0.7907]])
```

[15]:
```python
print(torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4]])) # Initialize from Python lists
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4]])
```

6 Common Tensor Operators

[16]:
```python
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()  # Assign a copy of `A` to `B` by allocating new memory
print(A)
print(A + B)
print(A * B)
print(A + 2)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([[ 0.,  2.,  4.,  6.],
        [ 8., 10., 12., 14.],
        [16., 18., 20., 22.],
        [24., 26., 28., 30.],
        [32., 34., 36., 38.]])
tensor([[  0.,   1.,   4.,   9.],
        [ 16.,  25.,  36.,  49.],
        [ 64.,  81., 100., 121.],
        [144., 169., 196., 225.],
        [256., 289., 324., 361.]])
```

```
tensor([[ 2.,  3.,  4.,  5.],
        [ 6.,  7.,  8.,  9.],
        [10., 11., 12., 13.],
        [14., 15., 16., 17.],
        [18., 19., 20., 21.]])
```

[17]:

```python
# Summations (analogously for A.mean())
print(A.sum())
print(A.sum(dim=0))
print(A.sum(dim=1))
```

```
tensor(190.)
tensor([40., 45., 50., 55.])
tensor([ 6., 22., 38., 54., 70.])
```

[18]:

```python
# Functions are applied elementwise
print(torch.exp(A))
print(A**2)
print(torch.pow(A, 2))
print(torch.cos(A))
```

```
tensor([[1.0000e+00, 2.7183e+00, 7.3891e+00, 2.0086e+01],
        [5.4598e+01, 1.4841e+02, 4.0343e+02, 1.0966e+03],
        [2.9810e+03, 8.1031e+03, 2.2026e+04, 5.9874e+04],
        [1.6275e+05, 4.4241e+05, 1.2026e+06, 3.2690e+06],
        [8.8861e+06, 2.4155e+07, 6.5660e+07, 1.7848e+08]])
tensor([[  0.,   1.,   4.,   9.],
        [ 16.,  25.,  36.,  49.],
        [ 64.,  81., 100., 121.],
        [144., 169., 196., 225.],
        [256., 289., 324., 361.]])
tensor([[  0.,   1.,   4.,   9.],
        [ 16.,  25.,  36.,  49.],
        [ 64.,  81., 100., 121.],
        [144., 169., 196., 225.],
        [256., 289., 324., 361.]])
tensor([[ 1.0000,  0.5403, -0.4161, -0.9900],
        [-0.6536,  0.2837,  0.9602,  0.7539],
        [-0.1455, -0.9111, -0.8391,  0.0044],
        [ 0.8439,  0.9074,  0.1367, -0.7597],
        [-0.9577, -0.2752,  0.6603,  0.9887]])
```

[19]:

```python
# Concatenation
print(torch.cat((A, B), dim=0))
print(torch.cat((A, B), dim=1))
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
```

```
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.],
        [ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([[ 0.,  1.,  2.,  3.,  0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.,  8.,  9., 10., 11.],
        [12., 13., 14., 15., 12., 13., 14., 15.],
        [16., 17., 18., 19., 16., 17., 18., 19.]])
```

7 Dot Product

- Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, we will let $\mathbf{x}^T\mathbf{y}$ denote their *dot product*. The dot product is the sum of the products of corresponding elements:

$$\mathbf{x}^T\mathbf{y} = \sum_{i=1}^{d} x_i y_i = x_1 y_1 + x_2 y_2 + ... + x_d y_d$$

.

[20]:
```
x = torch.arange(4, dtype=torch.float32)
y = torch.ones(4, dtype=torch.float32)
print(x)
print(y)
print(x.dot(y))
```

```
tensor([0., 1., 2., 3.])
tensor([1., 1., 1., 1.])
tensor(6.)
```

- Dot products are useful in many different ways.
- Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, if the elements of $\mathbf{y}$ are non-negative and sum to one (that is, $\sum_{i=1}^{d} y_i = y_1 + y_2 + ... + y_d = 1$), then the dot product $\mathbf{x}^T\mathbf{y}$ is the weighted average of the elements in $\mathbf{x}$ by the elements in $\mathbf{y}$
- Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ of *length* one (that is, $\|\mathbf{x}\|_2 = \|\mathbf{y}\|_2 = 1$, as detailed below), the dot product $\mathbf{x}^T\mathbf{y}$ is the cosine of the angle between them.

8 Matrix-Vector Multiplication

- Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{x} \in \mathbb{R}^n$. Let us write $\mathbf{A}$ in terms of its *row vectors*:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix},$$

where each $\mathbf{a}_i^T \in \mathbb{R}^n$ is a row vector representing the i^{th} row of the matrix $\mathbf{A}$.

- $\mathbf{Ax}$ is the column vector of length m whose i^{th} element is $\mathbf{a}_i^T\mathbf{x}$:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{x} \\ \mathbf{a}_2^T\mathbf{x} \\ \vdots \\ \mathbf{a}_m^T\mathbf{x} \end{bmatrix}.$$

- Multiplication by $\mathbf{A} \in \mathbb{R}^{m\times n}$ maps vectors from $\mathbb{R}^n$ to $\mathbb{R}^m$.

[21]:
```
# Matrix-vector multiplication
print(A)
print(x)
print(torch.mv(A, x))
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([0., 1., 2., 3.])
tensor([ 14.,  38.,  62.,  86., 110.])
```

9 Matrix Multiplication

- Consider two matrices $\mathbf{A} \in \mathbb{R}^{n\times k}$ and $\mathbf{B} \in \mathbb{R}^{k\times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}.$$

- Denote by $\mathbf{a}_i^T \in \mathbb{R}^k$ the *row vector* corresponding to the i^{th} row of $\mathbf{A}$, and by $\mathbf{b}_j \in \mathbb{R}^k$ the *column vector* corresponding to the j^{th} column of $\mathbf{B}$, so that

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}.$$

- $\mathbf{AB} \in \mathbb{R}^{n\times m}$ is a matrix given by:

$$\mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{b}_1 & \mathbf{a}_1^T\mathbf{b}_2 & \cdots & \mathbf{a}_1^T\mathbf{b}_m \\ \mathbf{a}_2^T\mathbf{b}_1 & \mathbf{a}_2^T\mathbf{b}_2 & \cdots & \mathbf{a}_2^T\mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^T\mathbf{b}_1 & \mathbf{a}_n^T\mathbf{b}_2 & \cdots & \mathbf{a}_n^T\mathbf{b}_m \end{bmatrix}.$$

[22]:
```python
# Matrix(-matrix) multiplication
print(A)
B = torch.arange(12, dtype=torch.float32).reshape(4,3)
print(B)
C = torch.mm(A, B)
print(C)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.],
        [ 9., 10., 11.]])
tensor([[ 42.,  48.,  54.],
        [114., 136., 158.],
        [186., 224., 262.],
        [258., 312., 366.],
        [330., 400., 470.]])
```

10 Norms

- Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ be vectors and $\alpha \in \mathbb{R}$ be a scalar. A *norm* is a function f that maps a vector to a scalar. A *norm* must satisfy the following properties:
  1. $f(\mathbf{x}) \geq 0$.
  2. $f(\mathbf{x}+\mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y})$.
  3. $f(\alpha\mathbf{x}) = |\alpha| f(\mathbf{x})$.
  4. $\mathbf{x} = \mathbf{0} \Leftrightarrow f(\mathbf{x}) = 0$, where $\mathbf{0}$ denotes the zero vector.
- The L_2 *norm* $\|\cdot\|_2$ gives the square root of the sum of the squares of the elements of a vector:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{d} x_i^2} = \sqrt{x_1^2 + x_2^2 + ... + x_d^2},$$

  which is the length of the vector. Because of the importance of this norm, the subscript 2 is often omitted, so that $\|\mathbf{x}\| = \|\mathbf{x}\|_2$.
- The L_1 *norm* $\|\cdot\|_1$ gives the sum of the absolute values of the elements of a vector:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{d} |x_i| = |x_1| + |x_2| + ... + |x_d|.$$

- The L_i distance between vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ is defined through the L_i norm as $\|\mathbf{x} - \mathbf{y}\|_i$.

11 Broadcasting Mechanism

- Mathematically, elementwise operations between tensors require them to have the same shape.
- Conveniently, under certain conditions, PyTorch can interpret and perform elementwise operations between tensors of different shapes using the *broadcasting mechanism*:
  - First, one or both tensors are expanded by copying elements in order to create two tensors with the same shape.
  - Second, the elementwise operation is carried on the resulting tensors.
- We typically broadcast across a dimension where an array has length 1, such as in the following example.

[23]:
```
print('Tensors:')
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
print(a)
print(b)

print('Sum with broadcasting:')
print(a + b)

print('Expanded tensors:')
a_expanded = torch.cat((a, a), dim=1)
b_expanded = torch.cat((b, b, b), dim=0)
print(a_expanded)
print(b_expanded)

print('Sum with expansion:')
print(a_expanded + b_expanded)
```

```
Tensors:
tensor([[0],
        [1],
        [2]])
tensor([[0, 1]])
Sum with broadcasting:
tensor([[0, 1],
        [1, 2],
        [2, 3]])
Expanded tensors:
tensor([[0, 0],
        [1, 1],
        [2, 2]])
tensor([[0, 1],
        [0, 1],
        [0, 1]])
Sum with expansion:
tensor([[0, 1],
        [1, 2],
```

```
        [2, 3]])
```

[24]:
```
# Another example
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
print(A)
B1 = A.sum(dim=1);
B2 = A.sum(dim=1, keepdims=True);
print(B1)
print(B2)
print(B1.size())
print(B2.size())
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([ 6., 22., 38., 54., 70.])
tensor([[ 6.],
        [22.],
        [38.],
        [54.],
        [70.]])
torch.Size([5])
torch.Size([5, 1])
```

[25]:
```
A/B2 # A/B1 does not work
```

[25]:
```
tensor([[0.0000, 0.1667, 0.3333, 0.5000],
        [0.1818, 0.2273, 0.2727, 0.3182],
        [0.2105, 0.2368, 0.2632, 0.2895],
        [0.2222, 0.2407, 0.2593, 0.2778],
        [0.2286, 0.2429, 0.2571, 0.2714]])
```

12 Tensor Indexing and Slicing

- Elements in a tensor can be accessed by indices that start at zero
- The range `i:j` includes all the elements from index i up to index j, including the element at i but excluding the element at j (just like the function `range`)
- As in Python lists, a negative index can be used to access elements starting from the last element along a dimension (for example, -1 denotes the index of last element along a dimension)

[26]:
```
# Indexing and slicing an array
A = torch.arange(5)
print(A)
print(A[2:5])
```

```
print(A[-2])
print(A[:3]) # A[0:3]
print(A[2:]) # A[2:5]
print(A[:]) # A[0:5]
```

```
tensor([0, 1, 2, 3, 4])
tensor([2, 3, 4])
tensor(3)
tensor([0, 1, 2])
tensor([2, 3, 4])
tensor([0, 1, 2, 3, 4])
```

[27]:
```
# Indexing and slicing a matrix
X = torch.arange(20, dtype=torch.float32).reshape(5, 4)
print(X)
print(X[1:3, :])
print(X[1:3, :2])
print(X[-1, :])
print(X[-3:-1, :])
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([[ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
tensor([[4., 5.],
        [8., 9.]])
tensor([16., 17., 18., 19.])
tensor([[ 8.,  9., 10., 11.],
        [12., 13., 14., 15.]])
```

13 Saving Memory

- Running operations can cause new memory to be allocated to store the results.
- We do not want to allocate memory unnecessarily all the time.
  - In machine learning, we might have hundreds of megabytes of parameters (or more!)
- Where possible, we want to perform operations *in-place*.

[28]:
```
# In-place example
X = torch.arange(20, dtype=torch.float32).reshape(5, 4)
Y = 10*X
print(id(X), id(Y))
Y = Y + X # not in-place
print(id(X), id(Y))
Y += X # in-place
```

```
print(id(X), id(Y))
Y[:] = Y + X # in-place, overrides the content of the tensor Y with the tensor␣
 ↪`Y + X`
print(id(X), id(Y))
```

```
136511199190432 136511199184272
136511199190432 136511199190992
136511199190432 136511199190992
136511199190432 136511199190992
```

[29]:
```
# Warning: the assignment operator `=` does not copy data, it just assigns names
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A
B[0, :] = -1
print(A)

# The method `clone` creates a copy of a tensor
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()
B[0, :] = -1
print(A)
```

```
tensor([[-1., -1., -1., -1.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.],
        [12., 13., 14., 15.],
        [16., 17., 18., 19.]])
```

14 Converting Tensors to Other Objects

[30]:
```
# Converting to a NumPy array, or vice-versa
A = X.numpy()
print(type(A))
B = torch.tensor(A)
print(type(B))
```

```
<class 'numpy.ndarray'>
<class 'torch.Tensor'>
```

[31]:
```
# Converting a size-1 tensor to a Python scalar
a = torch.tensor([3.5])
print(a)
```

```
print(a.item())
print(float(a))
print(int(a))
```

```
tensor([3.5000])
3.5
3.5
3
```

15 Recommended reading

- Dive into Deep Learning: Chapters 1, 2.1, 2.2, and 2.3.

16 [Storing this notebook as a pdf]

[32]:
```
%%capture
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic

# Set the path to this notebook below (add \ before spaces). The output `pdf`␣
↪will be stored in the corresponding folder.
!jupyter nbconvert --to pdf /content/gdrive/My\ Drive/Colab\ Notebooks/nndl/
↪week_01/lecture/Tensor_Data_Processing.ipynb

# If having issues, save this notebook (File > Save) and restart the session␣
↪(Runtime > Restart session) before running this cell. To debug, remove the␣
↪first line (`%%capture`).
```