Data Manipulation

- Basic data structure used in Deep Learning is the n-dimensional array, which is also called the *tensor*.
- *Tensor class* is called `Tensor` in PyTorch and is similar to NumPy's `ndarray` with a few killer features.
    - First, GPU is well-supported to accelerate the computation
    - Second, the tensor class supports automatic differentiation.
- These properties make the tensor class suitable for Deep Learning.

In [2]:
```
# To start, we import `torch`. Note that it's called PyTorch, we should import `
torch` instead of `pytorch`.
import torch
```

In [3]:
```
print(torch.__version__)
```

```
1.6.0
```

Tensor

- A tensor represents a (possibly multi-dimensional) array of numerical values.
    - A 1D tensor corresponds (in math) to a *vector*.
    - A 2D tensor corresponds to a *matrix*.
    - Tensors with more than two axes do not have special mathematical names.

Vector

- In math notation, we will usually denote vectors as bold-faced, lower-cased letters (e.g., $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$).
- Column vectors is the default orientation of vectors. In math, a column vector $\mathbf{x}$ can be written as

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

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

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

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

In [5]:
```
# access the i-th element: x[i]
print(x[3])
```

```
tensor(3)
```

In [6]:
```
# Vector shape i.e. dimensionality
print(len(x), x.size(), x.shape, type(x.size()))
```

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

Matrices

- Matrices (i.e. 2D tensors) will be typically denoted with bold-faced, capital letters (e.g., $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$).
- In math notation, we use $\mathbf{A} \in \mathbb{R}^{m \times n}$ to express that the matrix $\mathbf{A}$ consists of m rows and n columns of real-valued scalars.
- Visually, we can illustrate any matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ as a table, where each element a_{ij} belongs to the i^{th} row and j^{th} column:

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

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

In [7]:
```python
# Reshape function: change the shape of a tensor without changing the number of elements or their values
A = torch.arange(20).reshape(5, 4)
A, A[2, 3], A[2][3]
```

Out[7]:
```
(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))
```

In [8]:
```python
# Matrix shape
print(len(A), A.size(), A.shape)
```

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

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

Out[9]:
```
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```

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

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

Tensors

In [11]:

```
# A 3D tensor
X = torch.arange(24).reshape(2, 3, -1)
X
```

Out[11]:

```
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]]])
```

Commonly-used Tensor Constuctors

In [33]:
```
torch.ones((2, 3, 4)) # with Ones
```

Out[33]:
```
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.]]])
```

In [34]:
```
torch.zeros(2, 3) # with Zeros
```

Out[34]:
```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
```

In [35]:
```
torch.randn(3, 4) # samples from a Gaussian distribution with mean 0 and std of
1
```

Out[35]:
```
tensor([[-0.0183, -0.5366, -0.2664,  1.6010],
        [-0.0387, -0.5942,  0.3364, -0.1642],
        [-0.4362, -0.1339, -1.8351, -0.6304]])
```

In [37]:
```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4]]) # From Python lists
```

Out[37]:
```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4]])
```

Common Tensor Operators

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

Out[16]:
```
(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.]]))
```

In [17]:
```python
# Summations (same applies for mean() function)
A.sum(), A.sum(dim=0), A.sum(dim=1)
```

Out[17]:
```
(tensor(190.), tensor([40., 45., 50., 55.]), tensor([ 6., 22., 38., 54., 70.
]))
```

In [18]:
```python
# Functions are applied element-wise
torch.exp(A), A**2, torch.pow(A, 2), torch.cos(A)
```

Out[18]:
```
(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]]))
```

In [19]:
```python
# Concatenation
torch.cat((A, B), dim=0), torch.cat((A, B), dim=1)
```

Out[19]:
```
(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.]]))
```

Dot Products

- One of the most fundamental operations.
- Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their *dot product* $\mathbf{x}^\top \mathbf{y}$ (or $\langle \mathbf{x}, \mathbf{y} \rangle$) is a sum over the products of the elements at the same position: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

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

Out[20]:
```
(tensor([0., 1., 2., 3.]), tensor([1., 1., 1., 1.]), tensor(6.))
```

- Dot products are useful in a wide range of contexts:
  1. Given features stored in vector $\mathbf{x} \in \mathbb{R}^d$ and model weights in vector $\mathbf{w} \in \mathbb{R}^d$, the *score* between features and model weights are given by $\mathbf{x}^\top \mathbf{w}$.
  2. When the weights are non-negative and sum to one (i.e., $\left(\sum_{i=1}^{d} w_i = 1\right)$), the dot product expresses a *weighted average*.
  3. After normalizing two vectors to have the unit length (to be defined below), the dot products express the cosine of the angle between them.

Matrix-Vector Products

- Recall $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{x} \in \mathbb{R}^n$. Let us write $\mathbf{A}$ in terms of its row vectors:
$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix},$$
where each $\mathbf{a}_i^\top \in \mathbb{R}^n$ is a row vector representing the i^th row of the matrix $\mathbf{A}$.
- $\mathbf{A}\mathbf{x}$ is a column vector of length m, whose i^th element is $\mathbf{a}_i^\top \mathbf{x}$:
$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}.$$
- Multiplication by $\mathbf{A} \in \mathbb{R}^{m \times n}$ projects vectors from $\mathbb{R}^n$ to $\mathbb{R}^m$.

In [21]:
```
A.shape, x.shape, torch.mv(A, x)
```

Out[21]:
```
(torch.Size([5, 4]), torch.Size([4]), tensor([ 14.,  38.,  62.,  86., 110.])
)
```

Matrix-Matrix Multiplication

- Asssume that we have 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 $\mathbf{a}_i^\top \in \mathbb{R}^k$ the row vector representing the i^{th} row of $\mathbf{A}$, and by $\mathbf{b}_j \in \mathbb{R}^k$ the column vector from the j^{th} column of $\mathbf{B}$. We write $\mathbf{A}$ and $\mathbf{B}$ as:

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

- Then the matrix product $\mathbf{C} \in \mathbb{R}^{n\times m}$ is:

$$\mathbf{C} = \mathbf{A}\mathbf{B} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}.$$

In [22]:
```python
# Matrix multiplication
B = torch.ones(4, 3)
A, B, torch.mm(A, B)
```

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

Norms

- Some of the most useful operators in linear algebra are *norms*.
- Informally, the norm of a vector tells us how *big* a vector is (0 is the minimum).

- A (vector) norm is a function f that maps a vector to a scalar, satisfying the following properties:

    1. $f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x})$.
    2. $f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y})$.
    3. $f(\mathbf{x}) \geq 0$.
    4. $\forall i, x_i = 0 \Leftrightarrow f(\mathbf{x}) = 0$.

- The L_2 *norm* of $\mathbf{x}$ is the square root of the sum of the squares of the vector elements:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2},$$

where the subscript 2 is often omitted in L_2 norms, i.e., $\|\mathbf{x}\|$ is equivalent to $\|\mathbf{x}\|_2$.

- The L_1 *norm* is expressed as the sum of the absolute values of the vector elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| .$$

- In Deep Learning, we are often trying to solve optimization problems: e.g. *minimize* the distance between the model's predictions and the ground-truth observations.
    - The optimization obectives are ofter expressed as norms.

Broadcasting Mechanism

- In maths, in order to to perform elementwise operations between two tensors, they need to have the same shape.
- In Python and PyTorch, under certain conditions, even when their shapes differ, we can still perform elementwise operations by invoking the *broadcasting mechanism*.

- This mechanism works in the following way:

    - First, expand one or both arrays by copying elements appropriately so that after this transformation, the two tensors have the same shape.
    - Second, carry out the elementwise operations on the resulting arrays.

- In most cases, we broadcast along a dimension where an array initially only has length 1, such as in the following example.

In [23]:
```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b, a.size(), b.size()
```

Out[23]:
```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]),
 torch.Size([3, 1]),
 torch.Size([1, 2]))
```

In [24]:
```
a + b
```

Out[24]:
```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

In [25]:
```python
# Another example with sums
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
```

In [26]:
```python
B1 = A.sum(dim=1);
B2 = A.sum(dim=1, keepdims=True);
B1, B2
```

Out[26]:
```
(tensor([ 6., 22., 38., 54., 70.]),
 tensor([[ 6.],
         [22.],
         [38.],
         [54.],
         [70.]]))
```

In [27]:
```python
# Dimensionality
B1.size(), B2.size()
```

Out[27]:
```
(torch.Size([5]), torch.Size([5, 1]))
```

In [28]:
```python
A/B2 # A/B1 won't work
```

Out[28]:
```
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]])
```

Tensor Indexing and Slicing

- Just as in any other Python array, elements in a tensor can be accessed by index.
  - The first element has index 0 and ranges are specified to include the first but *before* the last element.
  - As in standard Python lists, we can access elements according to their relative position to the end of the list by using negative indices.
  - Example: `[-1]` selects the last element and `[1:3]` selects the second and the third elements as follows:

In [29]:
```
X = torch.arange(20, dtype=torch.float32).reshape(5, 4)
X, X[1:3, :], X[1:3, :2],  X[-1, :], X[-2:-1, :]
```

Out[29]:
```
(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([[12., 13., 14., 15.]]))
```

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
- Where possible, we want to perform these updates *in place*.

In [30]:

```
# 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
print(id(X), id(Y))
```

```
4906132176 4906163968
4906132176 4905804256
4906132176 4905804256
4906132176 4905804256
```

Conversion to Other Python Objects

In [31]:
```
# Converting to a NumPy array, or vice versa
A = X.numpy()
B = torch.tensor(A)
type(A), type(B)
```

Out[31]:
```
(numpy.ndarray, torch.Tensor)
```

In [32]:
```
# Converting to a size-1 tensor to a Python scalar,
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

Out[32]:
```
(tensor([3.5000]), 3.5, 3.5, 3)
```