en-ch4.2_Linear_Algebra

2019년 10월 8일

1 Chapter 4 - THE PRELIMINARIES: A CRASHCOURSE

1.1 4.2 Linear Algebra

```
In [2]: from mxnet import nd
```

4.2.1 Scalars

```
In [3]: x = nd.array([3.0])
        y = nd.array([2.0])

        print('x + y = ', x + y)
        print('x * y = ', x * y)
        print('x / y = ', x / y)
        print('x ** y = ', nd.power(x,y))
```

```
x + y =
[5.]
<NDArray 1 @cpu(0)>
x * y =
[6.]
<NDArray 1 @cpu(0)>
x / y =
[1.5]
<NDArray 1 @cpu(0)>
x ** y =
[9.]
<NDArray 1 @cpu(0)>
```

4.2.2 Vectors

```
In [4]: x = nd.arange(4)
        print('x = ', x)
```

```
x =
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
```

```
In [5]: x[3]
```

```
Out[5]:
        [3.]
        <NDArray 1 @cpu(0)>
```

4.2.3 Length, dimensionality and shape

```
In [6]: x.shape
```

```
Out[6]: (4,)
```

```
In [7]: a = 2
        x = nd.array([1,2,3])
        y = nd.array([10,20,30])

        print(a * x)
        print(a * x + y)
```

```
[2. 4. 6.]
<NDArray 3 @cpu(0)>

[12. 24. 36.]
<NDArray 3 @cpu(0)>
```

4.2.4 Matrices

```
In [8]: A = nd.arange(20).reshape((5,4))
        print(A)
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]]
<NDArray 5x4 @cpu(0)>
```

```
In [9]: print(A.T)
```

```
[[ 0.  4.  8. 12. 16.]
 [ 1.  5.  9. 13. 17.]
 [ 2.  6. 10. 14. 18.]
 [ 3.  7. 11. 15. 19.]]
<NDArray 4x5 @cpu(0)>
```

4.2.5 Tensors

- Just as vectors generalize scalars, and matrices generalize vectors, we can actually build data structures with even more axes. **Tensors give us a generic way of discussing arrays with an arbitrary number of axes.** Vectors, for example, are first-order tensors, and matrices are second-order tensors

```
In [10]: X = nd.arange(24).reshape((2, 3, 4))

         print('X.shape =', X.shape)
         print('X =', X)
```

```
X.shape = (2, 3, 4)
X =
[[[ 0.  1.  2.  3.]
```

```
  [ 4.  5.  6.  7.]
  [ 8.  9. 10. 11.]]

 [[12. 13. 14. 15.]
  [16. 17. 18. 19.]
  [20. 21. 22. 23.]]]
<NDArray 2x3x4 @cpu(0)>
```

4.2.6 Basic properties of tensor arithmetic

```
In [11]: a = 2
         x = nd.ones(3)
         y = nd.zeros(3)

         print(x.shape)
         print(y.shape)
         print((a * x).shape)
         print((a * x + y).shape)
```

```
(3,)
(3,)
(3,)
(3,)
```

- Scalars, vectors, matrices, and tensors of any order have some nice properties that we will often rely on. For example, ***as you might have noticed from the deinition of an element-wise operation, given operands with the same shape, the result of any element-wise operation is a tensor of that same shape.*** Another convenient property is that for **all tensors, multiplication by a scalar produces a tensor of the same shape.** In math, given two tensors X and Y with the same shape, aX + Y has the same shape (numerical mathematicians call this the AXPY operation).

4.2.7 Sums and means

- The next more sophisticated thing we can do with arbitrary tensors is to calculate the sum of their elements. In mathematical notation, we express sums using the $\sum$ symbol. To express

the sum of the elements in a vector $\mathbf{u}$ of length d, we can write $\sum_{i=1}^{d} u_i$. In code, we can write $\sum_{i=1}^{d} u_i$. In code, we can just call `nd.sum()`

```
In [12]: print(x)
         print(nd.sum(x))
```

```
[1. 1. 1.]
<NDArray 3 @cpu(0)>

[3.]
<NDArray 1 @cpu(0)>
```

- We can similarly express sums over the elements of tensors of arbitrary shape. For example, the sum of the elements of an $m \times n$ matrix A could be written $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
In [13]: print(A)
         print(nd.sum(A))
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]]
<NDArray 5x4 @cpu(0)>

[190.]
<NDArray 1 @cpu(0)>
```

- we could write the average over a vector $\mathbf{u}$ as $\frac{1}{d} \sum_{i=1}^{d} u_i$ and the average over a matrix A as $\frac{1}{n \cdot m} \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$. In code, we could just call `nd.mean()` on tensors of arbitrary shape:

```
In [14]: print(nd.mean(A))
         print(nd.sum(A) / A.size)
```

```
[9.5]
<NDArray 1 @cpu(0)>

[9.5]
<NDArray 1 @cpu(0)>
```

4.2.8 Dot products

- Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the dot product $\mathbf{u}^T\mathbf{v}$ is a sum over the products of the corresponding elements: $\mathbf{u}^T\mathbf{v} = \sum_{i=1}^{d} u_i \cdot v_i$.

```
In [15]: x = nd.arange(4)
         y = nd.ones(4)
         print(x, y, nd.dot(x, y))
```

```
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
[1. 1. 1. 1.]
<NDArray 4 @cpu(0)>
[6.]
<NDArray 1 @cpu(0)>
```

```
In [16]: nd.sum(x * y)
```

```
Out[16]:
         [6.]
         <NDArray 1 @cpu(0)>
```

4.2.9 Matrix-vector products

matrix A and a column vector $\mathbf{x}$.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

We can visualize the matrix in terms of its row vectors

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

Then the matrix vector product $\mathbf{y} = A\mathbf{x}$ is simply a column vector $\mathbf{y} \in \mathbb{R}^n$ where each entry y_i is the dot product $\mathbf{a}_i^T\mathbf{x}$.

$$A\mathbf{x} = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1^T\mathbf{x} \\ \mathbf{a}_2^T\mathbf{x} \\ \vdots \\ \mathbf{a}_n^T\mathbf{x} \end{pmatrix}$$

These transformations turn out to be remarkably useful. For example, we can represent rotations as multiplications by a square matrix. As we will see in subsequent chapters, we can also use matrix-vector products to describe the calculations of each layer in a neural network

```
In [17]: A.shape, x.shape
```

```
Out[17]: ((5, 4), (4,))
```

```
In [18]: nd.dot(A, x)
```

```
Out[18]:
         [ 14.  38.  62.  86. 110.]
         <NDArray 5 @cpu(0)>
```

4.2.10 Matrix-matrix multiplication Say we have two matrices, $A \in \mathbb{R}^{n\times k}$ and $B \in \mathbb{R}^{k\times m}$:

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

To produce the matrix product $C = AB$, it's easiest to think of A in terms of its row vectors and B in terms of its column vectors:

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

Note here that each row vector $\mathbf{a}_i^T$ lies in $\mathbb{R}^k$ and that each column vector $\mathbf{b}_j$ also lies in $\mathbb{R}^k$.

Then to produce the matrix product $C \in \mathbb{R}^{n \times m}$ we simply compute each entry c_{ij} as the dot product $\mathbf{a}_i^T \mathbf{b}_j$.

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

You can think of the matrix-matrix multiplication AB as simply performing m matrix-vector products and stitching the results together to form an $n \times m$ matrix. We can compute matrix-matrix products in PyTorch by using `nd.dot()`.

```
In [85]: B = nd.ones(shape=(4, 3))
         A.shape, B.shape, nd.dot(A, B)
```

```
Out[85]: ((5, 4), (4, 3),
          [[ 6.  6.  6.]
           [22. 22. 22.]
           [38. 38. 38.]
           [54. 54. 54.]
           [70. 70. 70.]]
          <NDArray 5x3 @cpu(0)>)
```

4.2.11 Norms `Norms` - they tell us how big a vector or matrix is.

- In fact, the Euclidean distance $\sqrt{x_1^2 + \cdots + x_n^2}$ is a norm. Specifically it is the ℓ_2-norm. An analogous computation, performed over the entries of a matrix, e.g. $\sqrt{\sum_{i,j} a_{ij}^2}$, is called the Frobenius norm. More often, in machine learning we work with the squared ℓ_2 norm (notated ℓ_2^2).

```
In [88]: # L2 norm
         x, nd.norm(x)
```

```
Out[88]: (
          [0. 1. 2. 3.]
          <NDArray 4 @cpu(0)>,
          [3.7416573]
          <NDArray 1 @cpu(0)>)

In [87]: # L1 norm
         nd.sum(nd.abs(x))

Out[87]:
         [6.]
         <NDArray 1 @cpu(0)>
```

4.2.12 Norms and objectives

- While we do not want to get too far ahead of ourselves, we do want you to anticipate why these concepts are useful. In machine learning we are often trying to solve optimization problems: Maximize the probability assigned to observed data. Minimize the distance between predictions and the ground-truth observations. Assign vector representations to items (like words, products, or news articles) such that the distance between similar items is minimized, and the distance between dissimilar items is maximized. Oftentimes, these objectives, ***perhaps the most important component of a machine learning algorithm (besides the data itself), are expressed as norms.***

4.2.13 Intermediate linear algebra

Basic vector properties

- ***Additive axioms*** (we assume that x,y,z are all vectors): $x + y = y + x$ and $(x + y) + z = x + (y + z)$ and $0 + x = x + 0 = x$ and $(-x) + x = x + (-x) = 0$.

- ***Multiplicative axioms*** (we assume that x is a vector and a, b are scalars): $0 \cdot x = 0$ and $1 \cdot x = x$ and $(ab)x = a(bx)$.

- ***Distributive axioms*** (we assume that x and y are vectors and a, b are scalars): $a(x + y) = ax + ay$ and $(a + b)x = ax + bx$.

Special matrices

- ***Symmetric Matrix*** $M^\top = M$
- ***Antisymmetric Matrix*** These matrices satisfy $M^\top = -M$ example)

$$M = \begin{pmatrix} 0 & 2 & -1 \\ -2 & 0 & -4 \\ 1 & 4 & 0 \end{pmatrix}$$

- ***Diagonally Dominant Matrix*** $M_{ii} \geq \sum_{j \neq i} M_{ij}$ example)

$$M = \begin{pmatrix} -4 & 2 & 1 \\ 1 & 6 & 2 \\ 1 & -2 & 5 \end{pmatrix}$$

- ***Positive Definite Matrix*** These are matrices that have the nice property where $x^\top Mx > 0$ whenever $x \neq 0$. Intuitively, they are a generalization of the squared norm of a vector $\|x\|^2 = x^\top x$. It is easy to check that whenever $M = A^\top A$, this holds since there $x^\top Mx = x^\top A^\top Ax = \|Ax\|^2$. example) $M = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix}$ $z^\top Mz = (z^\top M)z =$ $\begin{pmatrix} (2a-b) & (-1+2b-c) & (-b+2c) \end{pmatrix}$ $\begin{pmatrix} a \\ b \\ c \end{pmatrix} = 2a^2 - 2ab + 2b^2 - 2bc + 2c^2 = a^2 + (a - b)^2 + (b-c)^2 + c^2$
- If you are eager to learn more about linear algebra, here are some of our favorite resources on the topic
  - For a solid primer on basics, check out Gilbert Strang's book Introduction to Linear Algebra
  - Zico Kolter's Linear Algebra Review and Reference
