Linear algebra

```
In [1]: from mxnet import nd
```

In MXNet scalars are NDArrays with just one element.

In [2]:
```
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)>
```

We can convert any NDArray to a Python float by calling its `asscalar` method. Note that this is typically a bad idea. **While you are doing this, NDArray has to stop doing anything else in order to hand the result and the process control back to Python.** And unfortunately isn't very good at doing things in parallel.

In [3]:
```
print(x)
print(x.asscalar())
```

```
[3.]
<NDArray 1 @cpu(0)>
3.0
```

Vectors

Vectors are e.g. `[1.0,3.0,4.0,2.0]`. We use 1D NDArrays.

In [4]:
```
x = nd.arange(5)
print('x = ', x)
```

```
x = 
[0. 1. 2. 3. 4.]
<NDArray 5 @cpu(0)>
```

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

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

Length, dimensionality and shape

The length of a vector is commonly called its $dimension$. As with an ordinary Python array, we can access the length of an NDArray by calling Python's in-built `len()` function.

We can also access a vector's length via its `.shape` attribute. The shape is a tuple that lists the dimensionality along each of its axes.

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

```
Out[6]: (5,)
```

The word dimension is overloaded between number of axes and number of elements. **To avoid confusion, when we say *2D* array or *3D* array, we mean an array with 2 or 3 axes respectively. But if we say *n-dimensional* vector, we mean a vector of length $\mathbf{n}$.**

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)>
```

Matrices

Just as vectors generalize scalars from order 0 to order 1, matrices generalize vectors from $1D$ to $2D$. Matrices, which we'll typically denote with capital letters (A, B, C), are represented in code as arrays with 2 axes. Visually, we can draw a matrix as a table, where each entry a_{ij} belongs to the i-th row and j-th column.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

In [8]:
```
print(nd.arange(10))
A = nd.arange(20).reshape((5,4))
print(A)
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
<NDArray 10 @cpu(0)>

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

We can access elements a_{ij} by specifying row i and column j. Leaving them blank selects via `:` takes all ements in the respective dimension.

We can transpose the matrix through `T`. That is, if $B = A^T$, then $b_{ij} = a_{ji}$ for any i and j.

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)>
```

Tensors

Just as vectors generalize scalars, and matrices generalize vectors, we can increase the number of axes. When working with images the axes correspond to the height, width, and the three (RGB) color channels.

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)>
```

Basic properties of tensor arithmetic

Given two tensors X and Y with the same shape, $\alpha X + Y$ has the same shape (numerical mathematicians call this the AXPY operation).

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,)
```

Sums and means

In math 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 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)>
```

A related quantity is the *mean*. We calculate the mean by dividing the sum by the total number of elements. In code this is `nd.mean()`.

$$\mathrm{mean}(\mathbf{u}) = \frac{1}{d}\sum_{i=1}^{d} u_i \text{ and } \mathrm{mean}(A) = \frac{1}{n \cdot m}\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}$$

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

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

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

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)>
```

Note that we can express the dot product of two vectors `nd.dot(u, v)` equivalently by performing an element-wise multiplication and then a sum:

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

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

Matrix-vector products

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

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

So you can think of multiplication by a matrix $A \in \mathbb{R}^{m\times n}$ as a transformation that projects vectors from $\mathbb{R}^m$ to $\mathbb{R}^n$.

We can also use matrix-vector products to describe the calculations of each layer in a neural network. Expressing matrix-vector products in code with `ndarray`, we use the same `nd.dot()` function as for dot products.

```
In [17]: nd.dot(A, x)
```

```
Out[17]: [  14.   38.   62.   86.  110.]
         <NDArray 5 @cpu(0)>
```

Matrix-matrix multiplication

If you've gotten the hang of dot products and matrix-vector multiplication, then matrix-matrix multiplications should be pretty straightforward.

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

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

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

You can think of the matrix-matrix multiplication AB as simply performing m matrix-vector products and stitching the results together.

```
In [18]: B = nd.ones(shape=(4, 3))
         nd.dot(A, B)

Out[18]: [[ 6.  6.  6.]
          [22. 22. 22.]
          [38. 38. 38.]
          [54. 54. 54.]
          [70. 70. 70.]]
         <NDArray 5x3 @cpu(0)>
```

Norms

All norms must satisfy a handful of properties:

1. $\|\alpha A\| = |\alpha| \|A\|$
2. $\|A + B\| \leq \|A\| + \|B\|$
3. $\|A\| \geq 0$
4. If $\forall i, j, a_{ij} = 0$, then $\|A\| = 0$

To calculate the ℓ_2 norm, we can just call `nd.norm()`.

```
In [19]: nd.norm(x)
```

```
Out[19]: [3.7416573]
         <NDArray 1 @cpu(0)>
```

To calculate the ℓ_1-norm we can simply perform the absolute value and then sum over the elements.

```
In [20]: nd.sum(nd.abs(x))
```

```
Out[20]: [6.]
         <NDArray 1 @cpu(0)>
```