Lecture 2: Linear algebra done efficiently

CS4787/5777 — Principles of Large-Scale Machine Learning Systems

In [ ]:

In [ ]:

In [ ]:
```
import numpy
import scipy
import matplotlib
import time
```

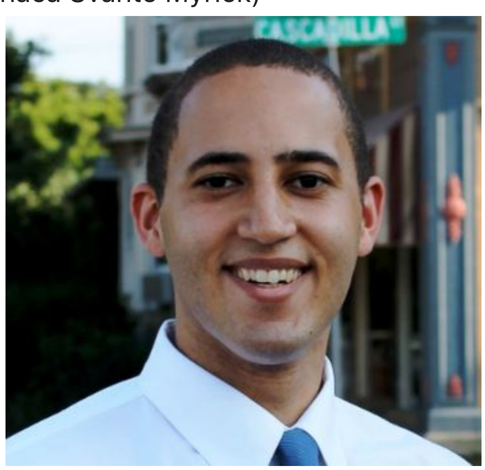
Recall our first principle from last lecture...

Principle #1: Write your learning task as an optimization problem, and solve it via fast algorithms that update the model iteratively with easy-to-compute steps using numerical linear algebra.

Recall our first principle from last lecture...

Principle #1: Write your learning task as an optimization problem, and solve it via fast algorithms that update the model iteratively with easy-to-compute steps using numerical linear algebra.

A simple example: we can represent the properties of an object using a **feature vector** (or embedding) in $\mathbb{R}^d$. Say we wanted to predict something about a group of people including this guy (former mayor of Ithaca Svante Myrick)



using the fact that he is 33, graduated in 2009, started being mayor in 2012, and makes $58,561 a year.

One way to represent this is as a vector in 4 dimensional space.

$$x = \begin{bmatrix} 36 \\ 2009 \\ 2012 \\ 58561 \end{bmatrix}.$$

Representing the information as a vector makes it easier for us to express ML models with it. We can then represent other objects we want to make predictions about with their own vectors, e.g.

$$x = \begin{bmatrix} 80 \\ 1965 \\ 2021 \\ 400000 \end{bmatrix}.$$

Linear Algebra: A Review

Before we start in on how to compute with vectors, matrices, et cetera, we should make sure we're all on the same page about what these objects are.

A vector (represented on a computer) is an array of numbers (usually floating point numbers). We say that the **dimension** (or length) of the vector is the size of the array, i.e. the number of numbers it contains.

A vector (in mathematics) is an element of a **vector space**. Recall: a vector space over the real numbers is a set V together with two binary operations $+$ (mapping $V \times V$ to V) and $\cdot$ (mapping $\mathbb{R} \times V$ to V) satisfying the following axioms for any $x, y, z \in V$ and $a, b \in \mathbb{R}$

- $x + y \in V$ and $a \cdot x = ax \in V$ *(closure)*
- $(x + y) + z = x + (y + z)$ *(associativity of addition)*
- $x + y = y + x$ *(transitivity of addition)*
- there exists a $(-x)$ such that $x + (-x) = 0$ *(negation)*
- $0 \in V$ such that $0 + x = x + 0 = x$ *(zero element)*
- $a(bx) = b(ax) = (ab)x$ *(associativity of scalar multiplication)*
- $1v = v$ *(multiplication by one)*
- $a(x + y) = ax + ay$ and $(a + b)x = ax + bx$ *(distributivity)*

We can treat our CS-style array of numbers as modeling a mathematical vector by letting $+$ add the two vectors elementwise and $\cdot$ multiply each element of the vector by the same scalar.

Again from the maths perspective, we say that a set of vectors $x_1, x_2, \ldots, x_d$ is **linearly independent** when no vector can be written as a linear combination of the others. That is,

$$\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_d x_d = 0 \;\leftrightarrow \alpha_1 = \alpha_2 = \cdots = \alpha_d = 0.$$

We say the **span** of some vectors $x_1, x_2, \ldots, x_d$ is the set of vectors that can be written as a linear combination of those vectors

$$\operatorname{span}(x_1, x_2, \ldots, x_d) = \{\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_d x_d \mid \alpha_i \in \mathbb{R}\}.$$

Finally, a set of vectors is a **basis** for the vector space V if it is linearly independent and if its span is the whole space V.

- Equivalently, a set of vectors is a basis if any vector $v \in V$ can be written uniquely as a linear combination of vectors in the basis.

We say the **dimension** of the space is d if it has a basis of size d.

What does this have to do with our computer-science definition of a vector?

If any vector v in the space can be written uniquely as

$$v = \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_d x_d$$

for some real numbers $\alpha_1, \alpha_2, \ldots$, then to represent v on a computer, it suffices to store α_1, α_2, $\ldots$, and α_d. We may as well store them in an array...and this gets us back to our CS-style notion of what a vector is.

- Importantly, this only works for finite-dimensional vector spaces!

Typically, when we work with a d-dimensonal vector space, we call it $\mathbb{R}^d$, and we use the **standard basis**, which I denote $e_1, \ldots, e_d$. E.g. in 3 dimensions this is defined as

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \; e_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \; e_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

and more generally e_i has a 1 in the ith entry of the vector and 0 otherwise. In this case, if x_i denotes the ith entry of a vector $x \in \mathbb{R}^d$, then

$$x = x_1 e_1 + x_2 e_2 + \cdots + x_d e_d = \sum_{i=1}^{d} x_i e_i.$$

In Python

In Python, we can use the library **numpy** to compute using vectors.

In [3]:
```
import numpy

u = numpy.array([1.0,2.0,3.0])
v = numpy.array([4.0,5.0,6.0])
```

```
print('u = {}'.format(u))
print('v = {}'.format(v))
print('u + v = {}'.format(u + v))
print('2 * u = {}'.format(2 * u))
```

```
u = [1. 2. 3.]
v = [4. 5. 6.]
u + v = [5. 7. 9.]
2 * u = [2. 4. 6.]
```

We can see that the standard vector operations are both supported easily!

Question: What have you seen represented as a vector in your previous experience with machine learning?

Answers:

- inputs/examples passed to the model
- feature vectors
- weights/parameters & bias
- outputs/predictions
- decision boundary hyperplane
- an image
- a gradient of the parameters
- word embeddings
- music/audio
- sensory input for robots
- physical position
- an attention matrix/vector

Linear Maps

We say a function F from a vector space U to a vector space V is a **linear map** if for any $x, y \in U$ and any $a \in \mathbb{R}$,

$$F(ax + y) = aF(x) + F(y).$$

- Notice that if we know $F(e_i)$ for all the basis elements e_i of U, then this uniquely determines F (why?).
- So, if we want to represent F on a computer and U and V are finite-dimensional vector spaces of dimensions m and n respectively, it suffices to store $F(e_1), F(e_2), \ldots, F(e_m)$.
- Each $F(e_i)$ is itself an element of V, which we can represent on a computer as an array of n numbers (since V is n-dimensional).
- So, we can represent F as an array of m arrays of n numbers...or equivalently as a **two-dimensional array**.

- Sadly, this overloads the meaning of the term "dimension"...but usually the meaning is clear from context.

Matrices

We call this two-dimensional-array representation of a linear map a **matrix**. Here is an example of a matrix in $\mathbb{R}^{3\times 3}$

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

We use multiplication to denote the effect of a matrix operating on a vector (this is equivalent to applying a multilinear map as a function). E.g. if F is the multilinear map corresponding to matrix A (really they are the same object, but I'm using different letters here to keep the notation clear), then

$$y = F(x) \;\equiv\; y = Ax.$$

We can add two matrices, and scale a matrix by a scalar.

- Note that this means that the set of matrices $\mathbb{R}^{n\times m}$ **is itself a vector space**.

Matrix Multiply

If $A \in \mathbb{R}^{n\times m}$ is the matrix that corresponds to the linear map F, and A_{ij} denotes the (i, j) th entry of the matrix, then by our construction

$$F(e_j) = \sum_{i=1}^{n} A_{ij} e_i$$

and so for any $x \in \mathbb{R}^m$

$$F(x) = F\left(\sum_{j=1}^{m} x_j e_j\right) = \sum_{j=1}^{m} x_j F(e_j) = \sum_{j=1}^{m} x_j \sum_{i=1}^{n} A_{ij} e_i = \sum_{i=1}^{n} \left(\sum_{j=1}^{m} A_{ij} x_j\right) e_i.$$

So, this means that the ith entry of $F(x)$ will be

$$(F(x))_i = \sum_{j=1}^{m} A_{ij} x_j.$$

Matrices in Python

A direct implementation of our matrix multiply formula:

$$(F(x))_i = \sum_{j=1}^{m} A_{ij} x_j.$$

In [4]:
```
x = numpy.array([1.0,2.0,3.0])
A = numpy.array([[1.0,2,3],[4,5,6]])

def matrix_multiply(A, x):
    (n,m) = A.shape
    assert(m == x.size)
    y = numpy.zeros(n)
    for i in range(n):
        for j in range(m):
            y[i] += A[i,j] * x[j]
    return y

print('x = {}'.format(x))
print('A = {}'.format(A))
print('Ax = {}'.format(matrix_multiply(A,x)))
```

```
x = [1. 2. 3.]
A = [[1. 2. 3.]
 [4. 5. 6.]]
Ax = [14. 32.]
```

In [5]:
```
# numpy has its own built-in support for matrix multiply
print('Ax = {}'.format(A @ x)) # numpy uses @ to mean matrix multiply
```

```
Ax = [14. 32.]
```

Using numpy buys us performance!

Comparing numpy matrix multiplies with my naive for-loop matrix multiply, one is much faster than the other.

In [12]:
```
# generate some random data
x = numpy.random.randn(1024)
A = numpy.random.randn(1024,1024)

import time
t = time.time()
for trial in range(20):
    B = matrix_multiply(A,x)

my_time = time.time() - t
print('my matrix multiply: {} seconds'.format(my_time))

t = time.time()
for trial in range(20):
    B = A @ x
np_time = time.time() - t
print('numpy matmul:       {} seconds'.format(np_time))

print('numpy was {:.0f}x faster'.format(my_time/np_time))
```

```
my matrix multiply: 4.296830177307129 seconds
numpy matmul:       0.0645897388458252 seconds
numpy was 67x faster
```

Question: What have you seen represented as a matrix in your previous experience with machine learning?

Answers:

- the weights in a layer of a neural network
- parameters
- tables
- a whole dataset
- image
- geometric transformation (physics, computer graphics)
- PCA
- covariance matrices
- Markov transition matrix
- Graph adjacency matrix & Graph Laplacian
- Hessian matrix

Multiplying Two Matrices

We can also multiply two matrices, which corresponds to function composition of linear maps.

- Of course, this only makes sense if the dimensions match!
- For example, if $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{q \times p}$, then it only makes sense to write AB if $m = q$.
- In this context, we often want to think of a vector $x \in \mathbb{R}^d$ as a $d \times 1$ matrix.

One special matrix is the **identity matrix** I, which has the property that $Ix = x$ for any x.

In [18]:
```
A = numpy.ones((2,3))
B = numpy.array([[3.0,8,1],[-7,2,-1],[0,2,-2]])
I = numpy.eye(3) # identity matrix

print('size of A = {}'.format(A.shape))
print('size of B = {}'.format(B.shape))

print('u = {}'.format(u))
print('A = {}'.format(A))
print('B = {}'.format(B))
print('I = {}'.format(I))
print('A.shape = {}'.format(A.shape))
print('B.shape = {}'.format(B.shape))
print('Iu = {}'.format(I @ u)) # numpy uses @ to mean matrix multiply
# print('Au = {}'.format(A @ u))
print('AB = {}'.format(A @ B))
print('BA = {}'.format(B @ A)) # should cause an error!
```

```
size of A = (2, 3)
size of B = (3, 3)
u = [1. 2. 3.]
A = [[1. 1. 1.]
 [1. 1. 1.]]
B = [[ 3.  8.  1.]
 [-7.  2. -1.]
 [ 0.  2. -2.]]
I = [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
A.shape = (2, 3)
B.shape = (3, 3)
Iu = [1. 2. 3.]
AB = [[-4. 12. -2.]
 [-4. 12. -2.]]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[18], line 17
     15 # print('Au = {}'.format(A @ u))
     16 print('AB = {}'.format(A @ B))
---> 17 print('BA = {}'.format(B @ A))

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, wi
th gufunc signature (n?,k),(k,m?)->(n?,m?) (size 2 is different from 3)
```

Transposition

Transposition takes a $n \times m$ matrix and **swaps the rows and columns** to produce an $m \times n$ matrix. Formally,

$$(A^T)_{ij} = A_{ji}.$$

A matrix that is its own transpose (i.e. $A = A^T$) is called a **symmetric matrix**.

We can also transpose a vector. Transposing a vector $x \in \mathbb{R}^d$ gives a matrix in $\mathbb{R}^{1 \times d}$, also known as a **row vector**. This gives us a handy way of defining the **dot product** which maps a pair of vectors to a scalar.

$$x^T y = y^T x = \langle x, y \rangle = \sum_{i=1}^{d} x_i y_i$$

- This is very useful in machine learning to express similarities, make predictions, compute norms, etc.
- It also gives us a handy way of grabbing the ith element of a vector, since $x_i = e_i^T x$ (and $A_{ij} = e_i^T A e_j$).
- A very useful identity: in $\mathbb{R}^d$, $\sum_{i=1}^{d} e_i e_i^T = I$.

In [20]:
```
A = numpy.array([[1,2,3],[4,5,6]])
print('A = {}'.format(A))
print('A.T = {}'.format(A.T))
```

```
print('u = {}'.format(u))
print('u.T @ u = {}'.format(u.T @ u))
```

```
A = [[1 2 3]
 [4 5 6]]
A.T = [[1 4]
 [2 5]
 [3 6]]
u = [1. 2. 3.]
u.T @ u = 14.0
```

Elementwise Operations

Often, we want to express some mathematics that goes beyond the addition and scalar multiplication operations in a vector space. Sometimes, to do this we use **elementwise operations** which operate on a vector/matrix (or pair of vectors/matrices) on a per-element basis. E.g. if

$$x = \begin{bmatrix} 1 \\ 4 \\ 9 \\ 16 \end{bmatrix},$$

then if sqrt operates elementwise,

$$\operatorname{sqrt}(x) = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}.$$

We can also do this with matrices and with binary operations.

Elementwise Operations in Python

In [21]:
```
x = numpy.array([1.0,4,9])
y = numpy.array([2,5,3])
z = numpy.array([2,3,7,8])

print('x = {}'.format(x))
print('y = {}'.format(y))
print('z = {}'.format(z))
print('sqrt(x) = {}'.format(numpy.sqrt(x)))
print('x * y = {}'.format(x * y)) # simple numerical operations are elementwise
print('x / y = {}'.format(x / y))
print('x * z = {}'.format(x * z)) # should cause error
```

```
x = [1. 4. 9.]
y = [2 5 3]
z = [2 3 7 8]
sqrt(x) = [1. 2. 3.]
x * y = [ 2. 20. 27.]
x / y = [0.5 0.8 3. ]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[21], line 11
      9 print('x * y = {}'.format(x * y)) # simple numerical operations are el
ementwise by default in numpy
     10 print('x / y = {}'.format(x / y))
---> 11 print('x * z = {}'.format(x * z))

ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

The Power of Broadcasting

We just saw that we can't use elementwise operations on pairs of vectors/matrices if they are not the same size. **Broadcasting** allows us to be more expressive by automatically expanding a vector/matrix along an axis of dimension 1.

In [53]:

```python
# x = numpy.array([2.0,3])
# A = numpy.array([[1.,2],[3,4]])

# print(x.shape)
# print(A.shape)

# print(x)
# print(A)

(numpy.ones((5,3,2)) @ (numpy.ones((7,2,4))))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[53], line 10
      1 # x = numpy.array([2.0,3])
      2 # A = numpy.array([[1.,2],[3,4]])
      3
   (...)
      7 # print(x)
      8 # print(A)
---> 10 (numpy.ones((5,3,2)) @ (numpy.ones((7,2,4))))

ValueError: operands could not be broadcast together with remapped shapes [ori
ginal->remapped]: (5,3,2)->(5,newaxis,newaxis) (7,2,4)->(7,newaxis,newaxis)  a
nd requested shape (3,4)
```

Tensors

We say that a matrix is stored as a 2-dimensional array. A tensor generalizes this to a matrix of whatever dimension you want.

From a mathematical perspective, a tensor is a **multilinear map** in the same way that a matrix is a linear map. That is, it's equivalent to a function

$$F(x_1, x_2, \ldots, x_n) \in \mathbb{R}$$

where F is linear in each of the inputs $x_i \in \mathbb{R}^{d_i}$ taken individually (i.e. with all the other inputs fixed).

$$F\left(\begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}, \begin{bmatrix} x_3 \\ y_3 \end{bmatrix}\right) = x_1 y_2 x_3.$$

We'll come back to this later when we discuss tensors in ML frameworks.

An Illustrative Example

Suppose that we have n websites, and we have collected a matrix $A \in \mathbb{R}^{n \times n}$, where A_{ij} counts the number of links from website i to website j.

We want to produce a new matrix $B \in \mathbb{R}^{n \times n}$ such that B_{ij} measures the *fraction* of links from website i that go to website j.

How do we compute this?

$$B_{ij} = \frac{A_{ij}}{\sum_{k=1}^{n} A_{ik}}$$

In [38]:
```
# generate some random data to work with
n = 6
A = numpy.random.randint(0,6,(n,n))**2 + numpy.random.randint(0,5,(n,n))

B_for = numpy.zeros((n,n))
for i in range(n):
    for j in range(n):
        acc = 0
        for k in range(n):
            acc += A[i,k]
        B_for[i,j] = A[i,j] / acc

# print(B_for - (A / numpy.sum(A, axis=1, keepdims=True)))

sumAik = A @ numpy.ones((n,1))
print(B_for - (A / sumAik))

# numpy.sum(A, axis=1).shape
```

```
[[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

Gradients

Many, if not most, machine learning training algorithms use gradients to optimize a function.

What is a gradient?

Suppose I have a function f from $\mathbb{R}^d$ to $\mathbb{R}$. The gradient, ∇f, is a function from $\mathbb{R}^d$ to $\mathbb{R}^d$ such that

$$(\nabla f(w))_i = \frac{\partial}{\partial w_i} f(w) = \lim_{\delta \to 0} \frac{f(w + \delta e_i) - f(w)}{\delta},$$

that is, it is the **vector of partial derivatives of the function**. Another, perhaps cleaner (and basis-independent), definition is that $\nabla f(w)^T$ is the linear map such that for any $u \in \mathbb{R}^d$

$$\nabla f(w)^T u = \lim_{\delta \to 0} \frac{f(w + \delta u) - f(w)}{\delta}.$$

More informally, it is the unique vector such that $f(w) \approx f(w_0) + (w - w_0)^T \nabla f(w_0)$ for w nearby w_0.

Let's derive some gradients!

$f(x) = x^T A x$

$f(x) = \|x\|_2^2 = \sum_{i=1}^d x_i^2$

...

$f(x) = \|x\|_1 = \sum_{i=1}^d |x_i|$

...

$f(x) = \|x\|_\infty = \max(|x_1|, |x_2|, \ldots, |x_d|)$

...

Takeaway: numpy gives us powerful capabilities to express numerical linear algebra...

...and you should become skilled in mapping from mathematical expressions to numpy and back.

In [ ]:
```

```

In [ ]:
```

```