Lecture 6:
Math Review II

Administrative

- HW0 due **tomorrow**, 1/29 11:59pm
- HW1 due **1 week from tomorrow**, 2/5 11:59pm

Last Time: Floating Point Math

IEEE 754 Single Precision / Single / float32

8 bits

$2^{127} \approx 10^{38}$

23 bits

$\approx$ 7 decimal digits

IEEE 754 Double Precision / Double / float64

11 bits

$2^{1023} \approx 10^{308}$

52 bits

$\approx$ 15 decimal digits

Last Time: Vectors

- Scale (vector, scalar → vector)
- Add (vector, vector → vector)
- Magnitude (vector → **scalar)**
- Dot product (vector, vector → **scalar)**
  - **Dot products are projection / angles**
- Cross product (vector, vector → **vector)**
  - Vectors facing same direction have cross product **0**
- You can **never** mix vectors of different sizes

Matrices

Matrices

Horizontally concatenate n, m-dim column vectors and you get a mxn matrix A (here 2x3)

$$\boldsymbol{A} = [\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n] = \begin{bmatrix} v_{1_1} & v_{2_1} & v_{3_1} \\ v_{1_2} & v_{2_2} & v_{3_2} \end{bmatrix}$$

a (scalar)
lowercase
undecorated

a (vector)
lowercase
bold or arrow

A (matrix)
uppercase
bold

Matrices

Horizontally concatenate n, m-dim column vectors and you get a mxn matrix A (here 2x3)

$$\boldsymbol{A} = [\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n] = \begin{bmatrix} \boxed{\begin{matrix} v_{1_1} \\ v_{1_2} \end{matrix}} & \boxed{\begin{matrix} v_{2_1} \\ v_{2_2} \end{matrix}} & \boxed{\begin{matrix} v_{3_1} \\ v_{3_2} \end{matrix}} \end{bmatrix}$$

Watch out: In math, it’s common to treat D-dim vector as a Dx1 matrix (column vector);
In numpy these are different things

Matrices

Transpose: flip rows / columns

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix}^T = \begin{bmatrix} a & b & c \end{bmatrix} \qquad (3\text{x}1)^{\text{T}} = 1\text{x}3$$

Vertically concatenate m, n-dim row vectors and you get a mxn matrix A (here 2x3)

$$A = \begin{bmatrix} \boldsymbol{u}_1^T \\ \vdots \\ \boldsymbol{u}_n^T \end{bmatrix} = \begin{bmatrix} \boxed{u_{1_1} \quad u_{1_2} \quad u_{1_3}} \\ \boxed{u_{2_1} \quad u_{2_2} \quad u_{2_3}} \end{bmatrix}$$

Matrix-vector Product

$$\boldsymbol{y}_{2x1} = \boldsymbol{A}_{2x3}\boldsymbol{x}_{3x1}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{v_1} & \boldsymbol{v_2} & \boldsymbol{v_3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\boldsymbol{y} = x_1\boldsymbol{v_1} + x_2\boldsymbol{v_2} + x_3\boldsymbol{v_3}$$

Linear combination of columns of ***A***

Matrix-vector Product

$$\boldsymbol{y}_{2x1} = \boldsymbol{A}_{2x3}\boldsymbol{x}_{3x1}$$

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \overset{\longleftarrow \boxed{3} \longrightarrow}{\begin{bmatrix} \quad \boldsymbol{u}_1^T \quad \\ \quad \boldsymbol{u}_2^T \quad \end{bmatrix}} \; \boldsymbol{x} \; \updownarrow \boxed{3}$$

$$y_1 = \boldsymbol{u}_1^T\boldsymbol{x} \qquad y_2 = \boldsymbol{u}_2^T\boldsymbol{x}$$

Dot product between rows of ***A*** *and* ***x***

Matrix Multiplication

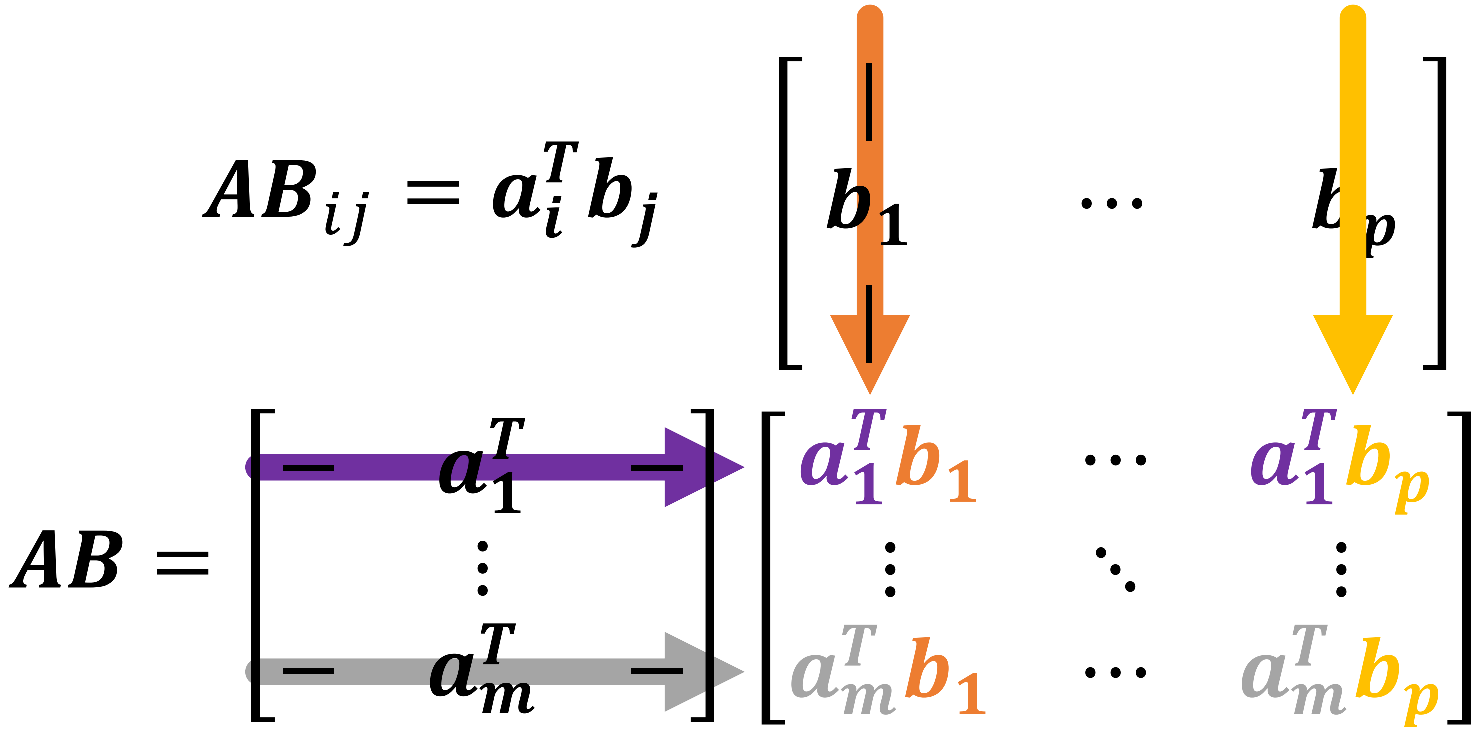
Generally: $\boldsymbol{A}_{mn}$ *and* $\boldsymbol{B}_{np}$ *yield product* $(\boldsymbol{AB})_{mp}$

$$\boldsymbol{AB} = \begin{bmatrix} - & \boldsymbol{a}_1^T & - \\ & \vdots & \\ - & \boldsymbol{a}_m^T & - \end{bmatrix} \begin{bmatrix} | & & | \\ \boldsymbol{b}_1 & \cdots & \boldsymbol{b}_p \\ | & & | \end{bmatrix}$$

Yes – in **A**, I’m referring to the rows, and in **B**, I’m referring to the columns

Matrix Multiplication

Generally: $\boldsymbol{A}_{mn}$ *and* $\boldsymbol{B}_{np}$ *yield product* $(\boldsymbol{AB})_{mp}$



$$\boldsymbol{AB}_{ij} = \boldsymbol{a}_i^T \boldsymbol{b}_j$$

$$\begin{bmatrix} | & & | \\ \boldsymbol{b}_1 & \cdots & \boldsymbol{b}_p \\ | & & | \end{bmatrix}$$

$$\boldsymbol{AB} = \begin{bmatrix} - & \boldsymbol{a}_1^T & - \\ & \vdots & \\ - & \boldsymbol{a}_m^T & - \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_1^T \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_1^T \boldsymbol{b}_p \\ \vdots & \ddots & \vdots \\ \boldsymbol{a}_m^T \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_m^T \boldsymbol{b}_p \end{bmatrix}$$

Matrix Multiplication

- Dimensions must match
- Dimensions must match
- Dimensions must match
- (Associative): ABx = (A)(Bx) = (AB)x
- (Not Commutative): ABx ≠ (BA)x ≠ (BxA)

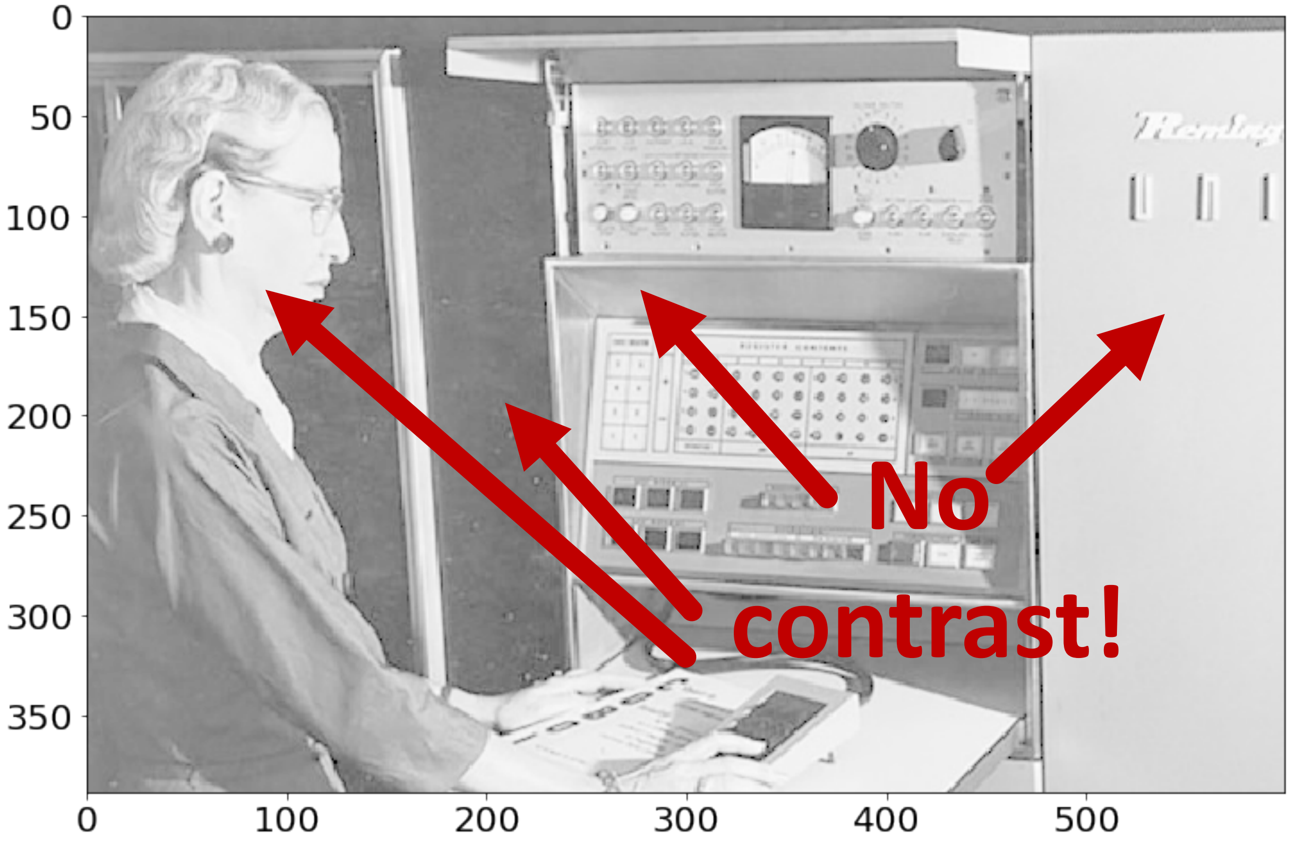
Two uses for Matrices

1. Storing things in a rectangular array (e.g. images)
    - *Typical operations*: element-wise operations, convolution (which we'll cover later)
    - *Atypical operations*: almost anything you learned in a math linear algebra class
2. A linear operator that maps vectors to another space (**Ax**)
    - *Typical/Atypical:* reverse of above

Images as Matrices

Suppose someone hands you this matrix.

What's wrong with it?



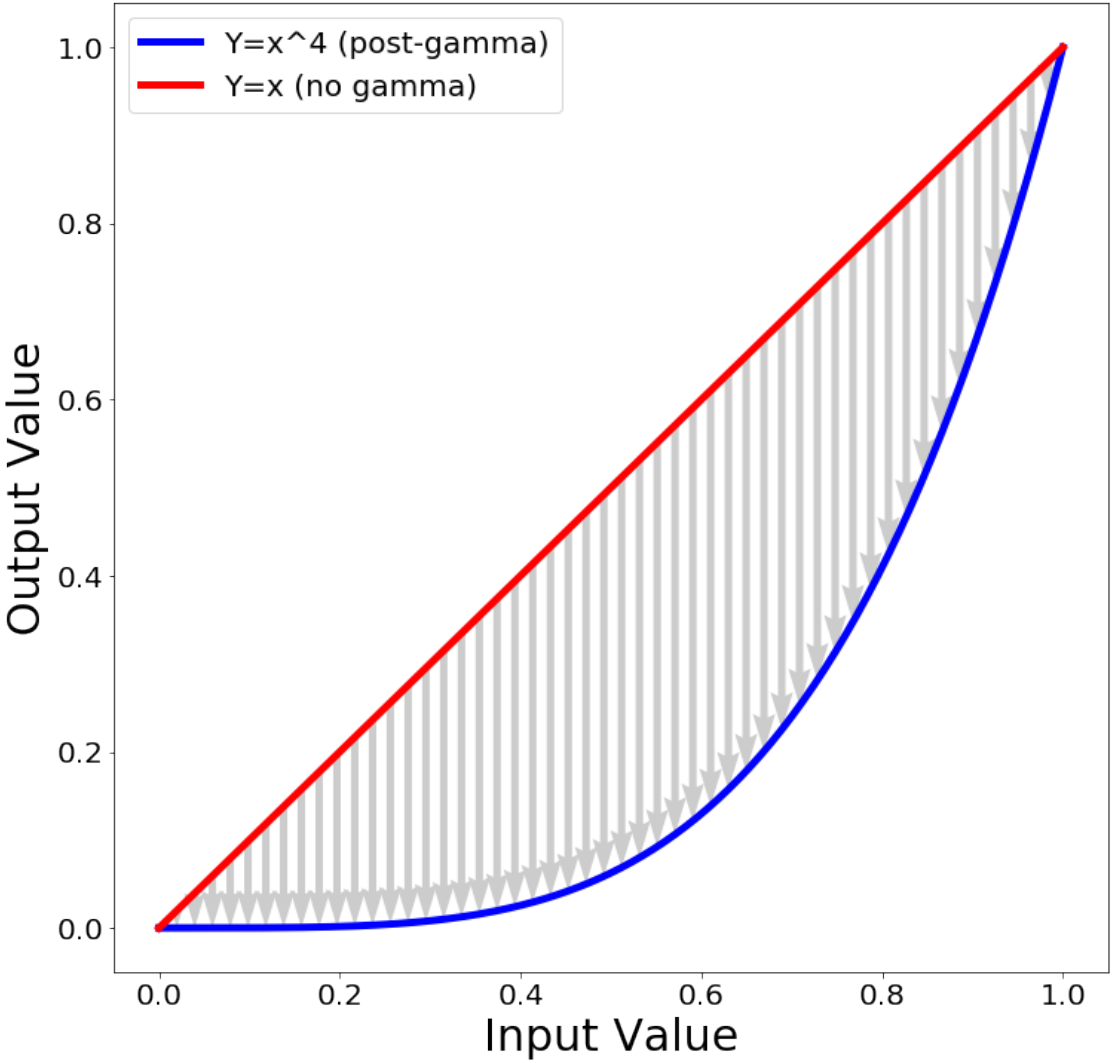
No contrast!

Contrast: Gamma Curve

Typical way to change the contrast is to apply a nonlinear correction

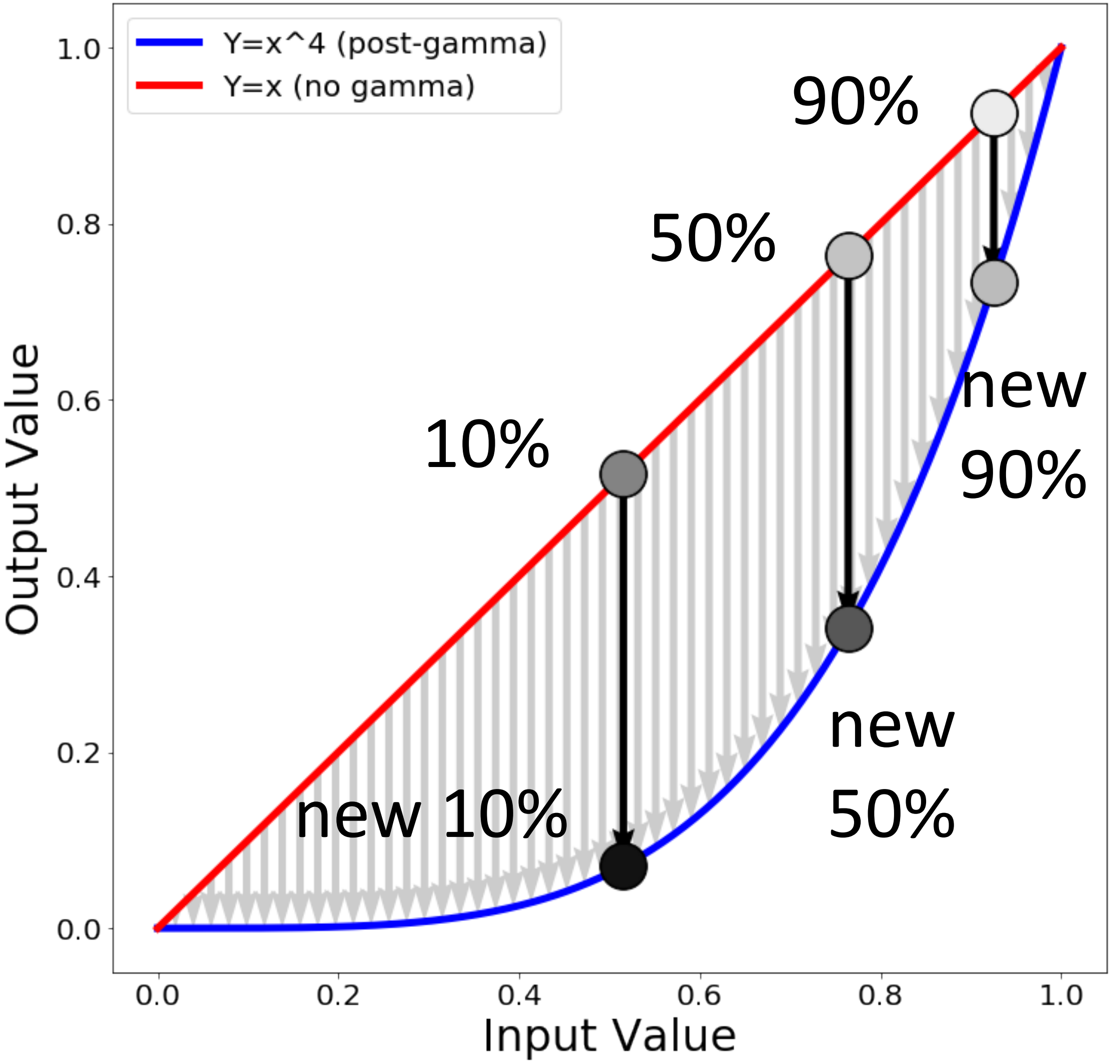
$$\text{pixelvalue}^{\gamma}$$

The quantity γ controls how much contrast gets added

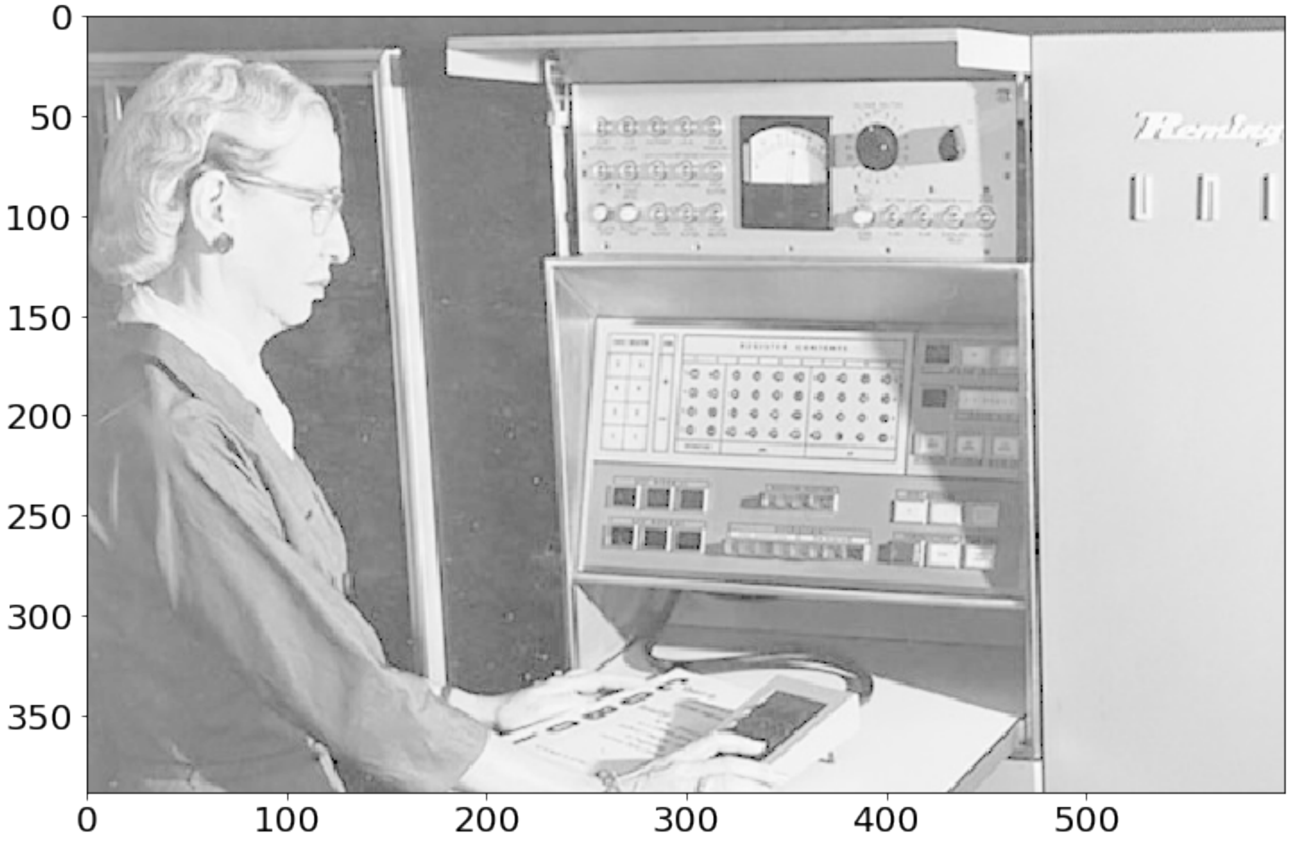


Contrast: Gamma Curve

Now the darkest regions (10th pctile) are **much** darker than the moderately dark regions (50th pctile).

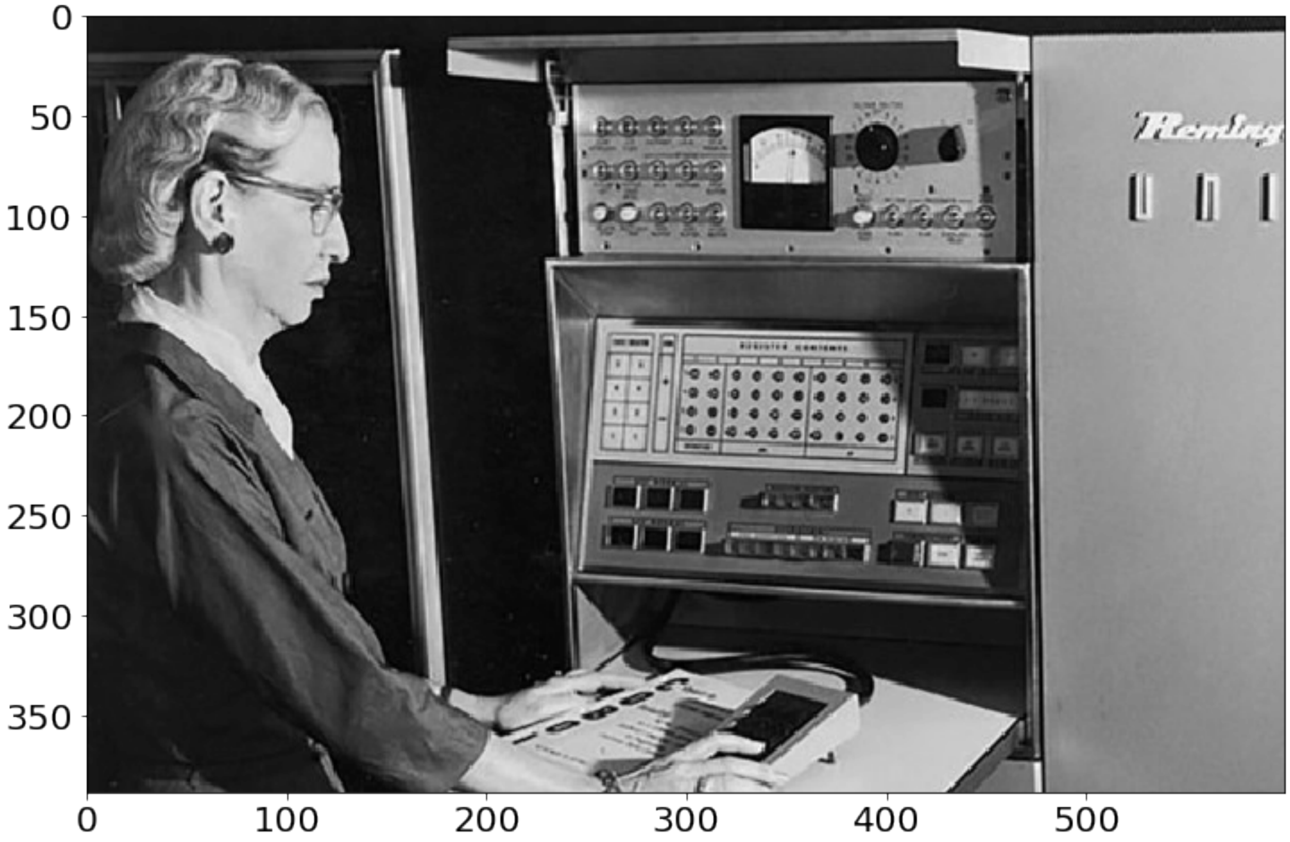


Contrast: Gamma Correction



Contrast: Gamma Correction

Phew! Much Better.



Implementation

Python+Numpy (right way):

```
imNew = im**4
```

Python+Numpy (slow way – **why?**):

```
imNew = np.zeros(im.shape)
for y in range(im.shape[0]):
  for x in range(im.shape[1]):
    imNew[y,x] = im[y,x]**expFactor
```

Elementwise Operations

Element-wise power – beware notation

$$(\boldsymbol{A}^p)_{ij} = A_{ij}^p$$

“Hadamard Product” / Element-wise multiplication

$$(\boldsymbol{A} \odot \boldsymbol{B})_{ij} = \boldsymbol{A}_{ij} * \boldsymbol{B}_{ij}$$

Element-wise division

$$(\boldsymbol{A}/\boldsymbol{B})_{ij} = \frac{A_{ij}}{B_{ij}}$$

Sums Across Axes

Suppose have Nx2 matrix **A**

$$\boldsymbol{A} = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix}$$

ND col. vec.

$$\Sigma(\boldsymbol{A}, 1) = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_n + y_n \end{bmatrix}$$

2D row vec

$$\Sigma(\boldsymbol{A}, 0) = \left[\sum_{i=1}^{n} x_i \quad , \sum_{i=1}^{n} y_i \right]$$

Note – libraries distinguish between N-D column vector and Nx1 matrix.

Operations they don't teach

You Probably Saw Matrix Addition

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix}$$

What is this? FYI: e is a scalar

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + e = \qquad \begin{bmatrix} a+e & b+e \\ c+e & d+e \end{bmatrix}$$

Broadcasting

If you want to be pedantic and proper, you expand e by multiplying a matrix of 1s (denoted **1**)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + e = \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \mathbf{1}_{2x2} e$$

$$= \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & e \\ e & e \end{bmatrix}$$

Many smart matrix libraries do this automatically. This is the source of many bugs.

Broadcasting Example

Given: a nx2 matrix **P** and a 2D column vector **v**, Want: nx2 difference matrix **D**

$$\boldsymbol{P} = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} \quad \boldsymbol{v} = \begin{bmatrix} a \\ b \end{bmatrix} \quad \boldsymbol{D} = \begin{bmatrix} x_1 - a & y_1 - b \\ \vdots & \vdots \\ x_n - a & y_n - b \end{bmatrix}$$

$$\boldsymbol{P} - \boldsymbol{v}^T = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} - \begin{matrix} [a & b] \\ \vdots & \\ [a & b] \end{matrix}$$

Blue stuff is assumed / broadcast

Broadcasting Rules

Suppose we have numpy arrays x and y.
How will they broadcast?

1. Write down the **shape** of each array as a tuple of integers:
For example: x: (10,) y: (20, 10)

2. If they have different numbers of dimensions, **prepend** with ones until they have the same number of dimensions
For example: x: (10,) y: (20, 10) → x: (1, 10) y: (20, 10)

3. Compare each dimension. There are 3 cases:
   (a) Dimension match. Everything is good
   (b) Dimensions don't match, but one is =1.
   "Duplicate" the smaller array along that axis to match
   (c) Dimensions don't match, neither are =1. Error!

Broadcasting Examples

```
x = np.ones(10, 20)
y = np.ones(20)
z = x + y
print(z.shape)
(10,20)
```

```
x = np.ones(10, 20)
y = np.ones(10)
z = x + y
print(z.shape)
ERROR
```

```
x = np.ones(10, 20)
y = np.ones(10, 1)
z = x + y
print(z.shape)
(10,20)
```

```
x = np.ones(1, 20)
y = np.ones(10, 1)
z = x + y
print(z.shape)
(10,20)
```

Tensors

Scalar: Just one number

Vector: 1D list of numbers

Matrix: 2D grid of numbers

Tensor: N-dimensional grid of numbers
(Lots of other meanings in math, physics)

Broadcasting with Tensors

The same broadcasting rules apply to tensors with any number of dimensions!

```
x = np.ones(30)
y = np.ones(20, 1)
z = np.ones(10, 1, 1)
w = x + y + z
print(w.shape)
```

```
(10, 20, 30)
```

Vectorization

Writing code without explicit loops: use broadcasting, matrix multiply, and other (optimized) numpy primitives instead

Vectorization Example

- Suppose I represent each image as a 128-dimensional vector
- I want to compute all the pairwise distances between {$\mathbf{x}_1$, …, $\mathbf{x}_N$} and {$\mathbf{y}_1$, …, $\mathbf{y}_M$} so I can find, for every $\mathbf{x}_i$ the nearest $\mathbf{y}_j$
- Identity: $||\boldsymbol{x} - \boldsymbol{y}||^2 = ||\boldsymbol{x}||^2 + ||\boldsymbol{y}||^2 - 2\boldsymbol{x}^T\boldsymbol{y}$
- Or: $||\boldsymbol{x} - \boldsymbol{y}|| = (||\boldsymbol{x}||^2 + ||\boldsymbol{y}||^2 - 2\boldsymbol{x}^T\boldsymbol{y})^{1/2}$

Vectorization Example

$$\boldsymbol{X} = \begin{bmatrix} - & \boldsymbol{x}_1 & - \\ & \vdots & \\ - & \boldsymbol{x}_N & - \end{bmatrix} \quad \boldsymbol{Y} = \begin{bmatrix} - & \boldsymbol{y}_1 & - \\ & \vdots & \\ - & \boldsymbol{y}_M & - \end{bmatrix} \quad \boldsymbol{Y}^T = \begin{bmatrix} | & & | \\ \boldsymbol{y}_1 & \cdots & \boldsymbol{y}_M \\ | & & | \end{bmatrix}$$

Compute a Nx1 vector of norms (can also do Mx1)

$$\boldsymbol{\Sigma}\left(\boldsymbol{X}^2, \boldsymbol{1}\right) = \begin{bmatrix} \|\boldsymbol{x}_1\|^2 \\ \vdots \\ \|\boldsymbol{x}_N\|^2 \end{bmatrix}$$

Compute a NxM matrix of dot products

$$\left(\boldsymbol{X}\boldsymbol{Y}^T\right)_{ij} = \boldsymbol{x}_i^T \boldsymbol{y}_j$$

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^{\boldsymbol{T}} - 2\boldsymbol{X}\boldsymbol{Y}^{\boldsymbol{T}}\right)^{1/2}$$

$$\begin{bmatrix} \|\boldsymbol{x}_1\|^2 \\ \vdots \\ \|\boldsymbol{x}_N\|^2 \end{bmatrix} + \begin{bmatrix} \|\boldsymbol{y}_1\|^2 & \cdots & \|\boldsymbol{y}_M\|^2 \end{bmatrix}$$

$$\begin{bmatrix} \|\boldsymbol{x}_1\|^2 + \|\boldsymbol{y}_1\|^2 & \cdots & \|\boldsymbol{x}_1\|^2 + \|\boldsymbol{y}_M\|^2 \\ \vdots & \ddots & \vdots \\ \|\boldsymbol{x}_N\|^2 + \|\boldsymbol{y}_1\|^2 & \cdots & \|\boldsymbol{x}_N\|^2 + \|\boldsymbol{y}_M\|^2 \end{bmatrix}$$

Why?

$$\left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^T\right)_{ij} = \|\boldsymbol{x}_i\|^2 + \left\|\boldsymbol{y}_j\right\|^2$$

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^{\boldsymbol{T}} - 2\boldsymbol{X}\boldsymbol{Y}^{\boldsymbol{T}}\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{y}_j\|^2 + 2\boldsymbol{x}^T\boldsymbol{y}$$

Numpy code:

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.
Suppose X is (N, D), Y is (M, D):

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^{\boldsymbol{T}} - 2\boldsymbol{X}\boldsymbol{Y}^{\boldsymbol{T}}\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{y}_j\|^2 + 2\boldsymbol{x}^T\boldsymbol{y}$$

(N, 1)

Numpy code:

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.
Suppose X is (N, D), Y is (M, D):

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^{\boldsymbol{T}} - 2\boldsymbol{X}\boldsymbol{Y}^{\boldsymbol{T}}\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{y}_j\|^2 + 2\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{y}$$

Numpy code: (N, 1) (M, 1)

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.
Suppose X is (N, D), Y is (M, D):

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^T - 2\boldsymbol{X}\boldsymbol{Y}^T\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \left\|\boldsymbol{y}_j\right\|^2 + 2\boldsymbol{x}^T\boldsymbol{y}$$

Numpy code: (N, 1) (M, 1) (N, M)

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.
Suppose X is (N, D), Y is (M, D):

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^T - 2\boldsymbol{X}\boldsymbol{Y}^T\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{y}_j\|^2 + 2\boldsymbol{x}^T\boldsymbol{y}$$

Numpy code: (N, 1) (M, 1) (N, M) (N, M)

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.
Suppose X is (N, D), Y is (M, D):

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^{\boldsymbol{T}} - 2\boldsymbol{X}\boldsymbol{Y}^{\boldsymbol{T}}\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{y}_j\|^2 + 2\boldsymbol{x}^T\boldsymbol{y}$$

Numpy code: (N, 1) (M, 1) (N, M) (N, M)

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples. Suppose X is (N, D), Y is (M, D):

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^{\boldsymbol{T}} - 2\boldsymbol{X}\boldsymbol{Y}^{\boldsymbol{T}}\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \|\boldsymbol{y}_j\|^2 + 2\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{y}$$

Numpy code: (N, 1) (M, 1) (N, M) (N, M)

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

Get in the habit of thinking about shapes as tuples.

Suppose X is (N, D), Y is (M, D):

Vectorization Example

$$\mathbf{D} = \left(\Sigma(\boldsymbol{X}^2, 1) + \Sigma(\boldsymbol{Y}^2, 1)^T - 2\boldsymbol{X}\boldsymbol{Y}^T\right)^{1/2}$$

$$\mathbf{D}_{ij} = \|\boldsymbol{x}_i\|^2 + \left\|\boldsymbol{y}_j\right\|^2 + 2\boldsymbol{x}^T\boldsymbol{y}$$

Numpy code:

```
XNorm = np.sum(X**2,axis=1,keepdims=True)
YNorm = np.sum(Y**2,axis=1,keepdims=True)
D = (XNorm+YNorm.T-2*np.dot(X,Y.T))**0.5
```

*May have to make sure this is at least 0 (sometimes roundoff issues happen)

Does Vectorization Matter?

Computing pairwise distances between 300 and 400 128-dimensional vectors

1. for x in X, for y in Y, using native python: 9s
2. for x in X, for y in Y, using numpy to compute distance: 0.8s
3. vectorized: 0.0045s (~2000x faster than 1, 175x faster than 2)

Expressing things in primitives that are optimized is usually faster

Even more important with special hardware like GPUs or TPUs!

Linear Algebra

Linear Independence

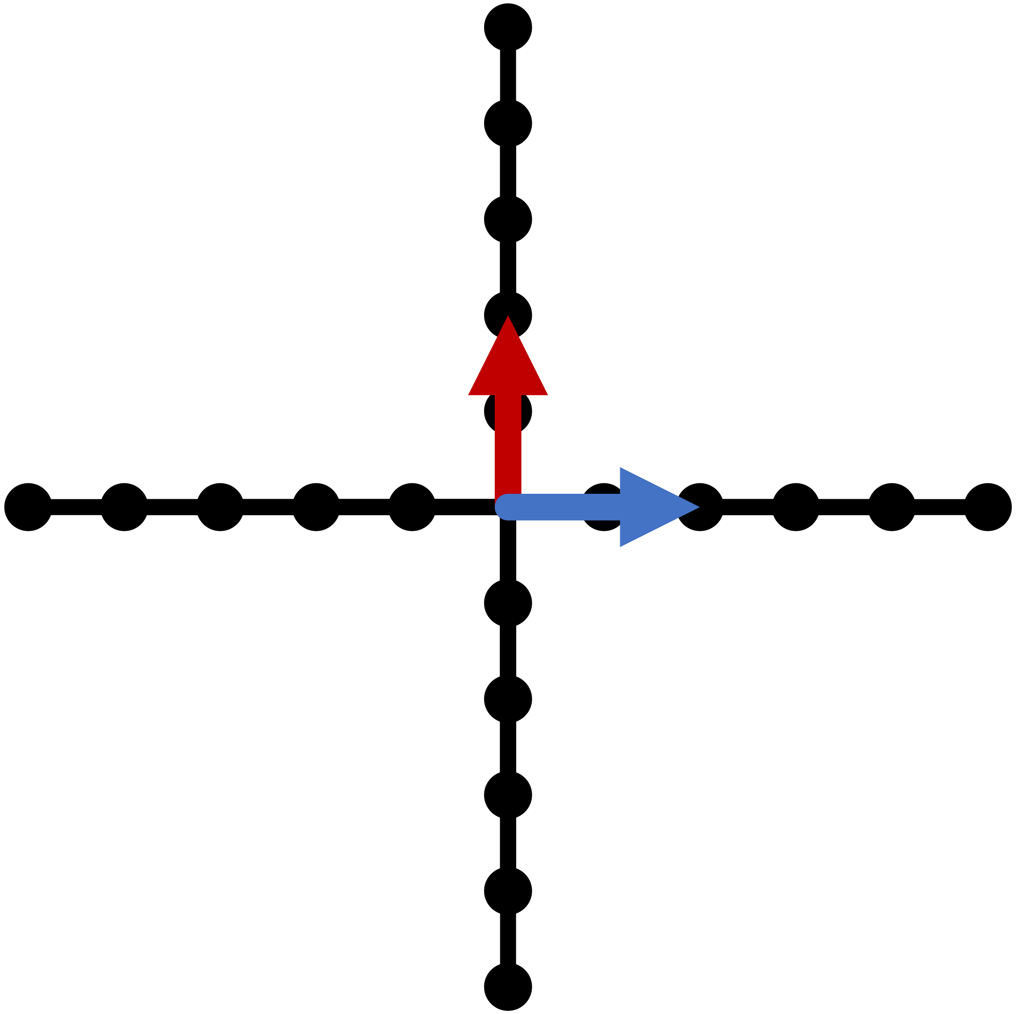
A set of vectors is **linearly independent** if you can't write one as a linear combination of the others.

Suppose: $\boldsymbol{a} = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$ $\boldsymbol{b} = \begin{bmatrix} 0 \\ 6 \\ 0 \end{bmatrix}$ $\boldsymbol{c} = \begin{bmatrix} 5 \\ 0 \\ 0 \end{bmatrix}$

$$\boldsymbol{x} = \begin{bmatrix} 0 \\ 0 \\ 4 \end{bmatrix} = \qquad \boldsymbol{y} = \begin{bmatrix} 0 \\ -2 \\ 1 \end{bmatrix} = \frac{1}{2}\boldsymbol{a} - \frac{1}{3}\boldsymbol{b}$$

- **Is the set {a,b,c} linearly independent?**
- **Is the set {a,b,x} linearly independent?**
- **Max # of independent 3D vectors?**

Span



Span: all linear combinations of a set of vectors

Span({ ↑ }) =

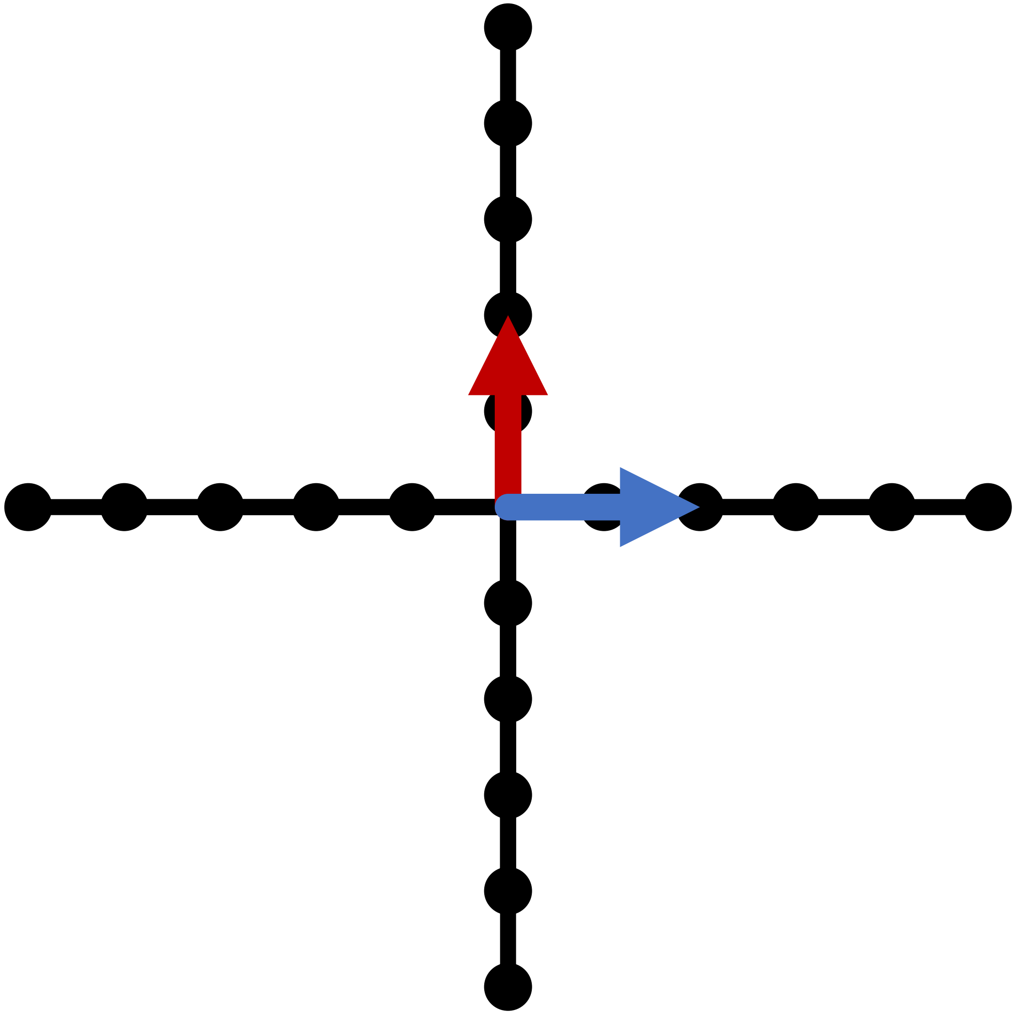
Span({[0,2]}) = ?

All vertical lines through origin =

$\{\lambda[0,1]: \lambda \in R\}$

Is blue in {red}'s span?

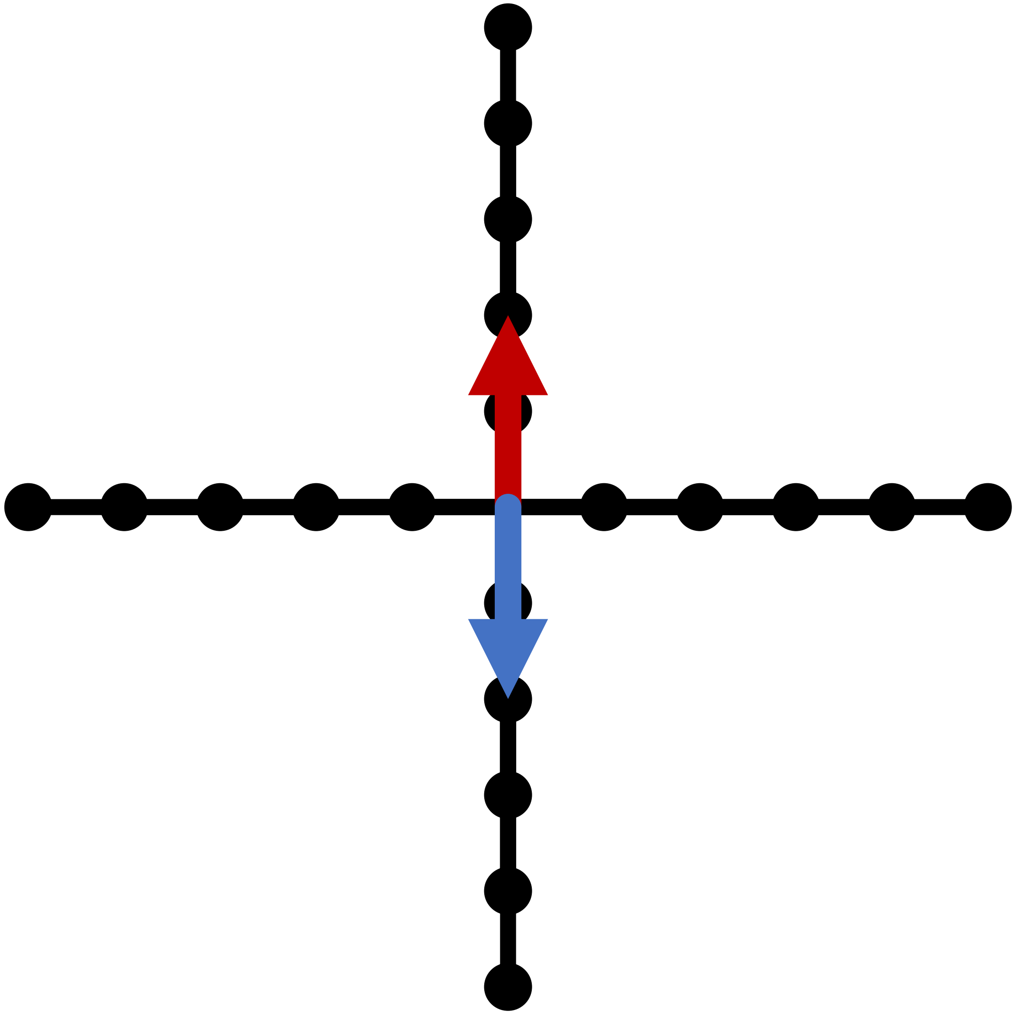
Span



Span: all linear combinations of a set of vectors

Span({↑ , →}) = **?**

Span



Span: all linear combinations of a set of vectors

Span({↑ , ↓ }) = **?**

Matrix-Vector Product

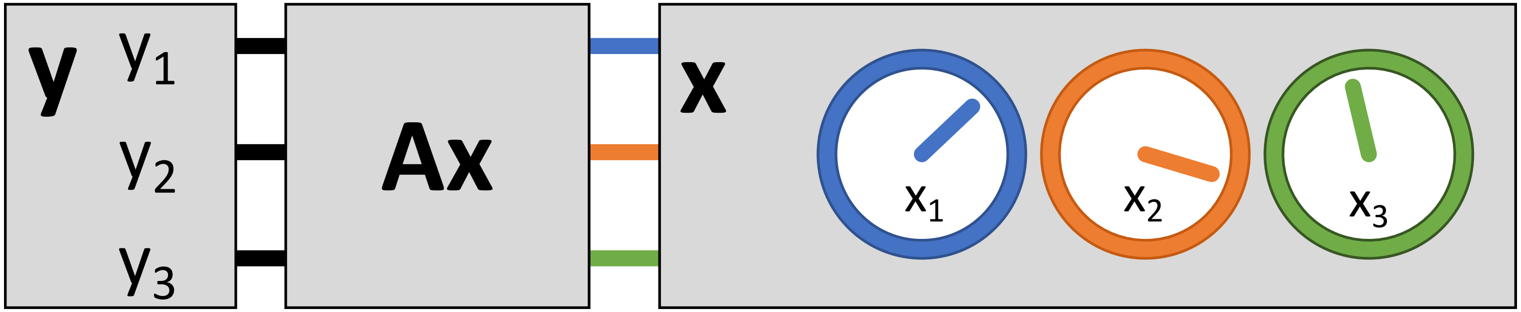
$$\boldsymbol{Ax} = \begin{bmatrix} | & & | \\ \boldsymbol{c_1} & \cdots & \boldsymbol{c_n} \\ | & & | \end{bmatrix} \boldsymbol{x}$$

Right-multiplying **A** by **x** mixes columns of **A** according to entries of **x**

- The output space of f(**x**) = **Ax** is constrained to be the *span* of the columns of **A**.
- Can’t output things you can’t construct out of your columns

An Intuition

$$\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x} = \begin{bmatrix} | & | & | \\ \boldsymbol{c}_1 & \boldsymbol{c}_2 & \boldsymbol{c}_n \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



x – knobs on machine (e.g., fuel, brakes)

y – state of the world (e.g., where you are)

A – machine (e.g., your car)

Linear Independence

Suppose the columns of 3x3 matrix **A** are *not* linearly independent (c_1, αc_1, c_2 for instance)

$$\boldsymbol{y} = \boldsymbol{Ax} = \begin{bmatrix} | & | & | \\ \boldsymbol{c_1} & \alpha\boldsymbol{c_1} & \boldsymbol{c_2} \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

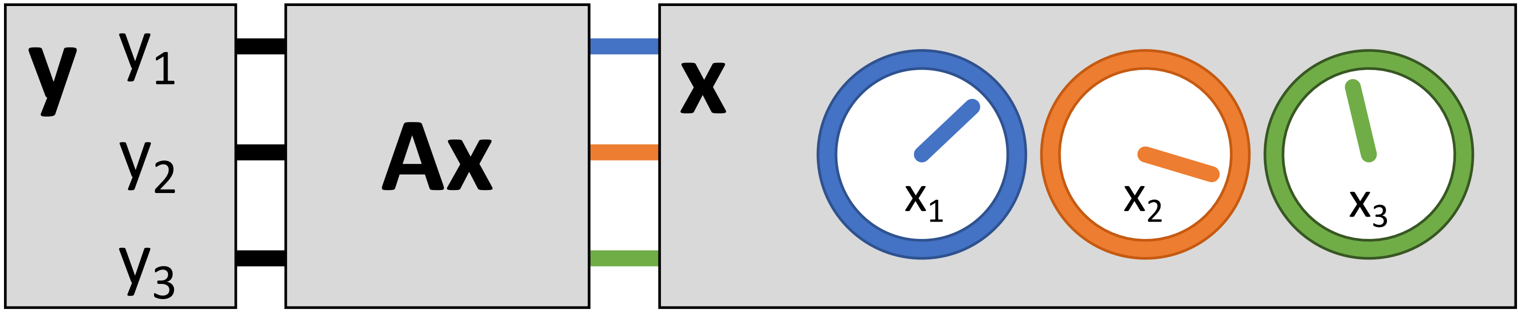
$$\boldsymbol{y} = x_1\boldsymbol{c_1} + \alpha x_2\boldsymbol{c_1} + x_3\boldsymbol{c_2}$$

$$\boldsymbol{y} = (x_1 + \alpha x_2)\boldsymbol{c_1} + x_3\boldsymbol{c_2}$$

Linear Independence Intuition

Knobs of **x** are redundant. Even if **y** has 3 outputs, you can only control it in two directions

$$\boldsymbol{y} = (x_1 + \alpha x_2)\boldsymbol{c_1} + x_3 \boldsymbol{c_2}$$



Linear Independence

Recall: $\boldsymbol{Ax} = (x_1 + \alpha x_2)\boldsymbol{c_1} + x_3 \boldsymbol{c_2}$

$$\boldsymbol{y} = \boldsymbol{A}\begin{bmatrix} x_1 + \beta \\ x_2 - \beta/\alpha \\ x_3 \end{bmatrix} = \left(x_1 + \cancel{\beta} + \alpha x_2 - \cancel{\alpha \frac{\beta}{\alpha}}\right) c_1 + x_3 c_2$$

- Can write **y** an infinite number of ways by adding β to $\mathbf{x_1}$ and subtracting $\frac{\beta}{\alpha}$ from $\mathbf{x_2}$
- Or, given a vector **y** there's not a unique vector **x** s.t. **y** =**Ax**
- Not all **y** have a corresponding **x** s.t. **y=Ax** (assuming $\boldsymbol{c_1}$ and $\boldsymbol{c_1}$ have dimension >= 3)

Linear Independence

$$\boldsymbol{Ax} = (x_1 + \alpha x_2)\boldsymbol{c_1} + x_3\boldsymbol{c_2}$$

$$\boldsymbol{y} = \boldsymbol{A}\begin{bmatrix} \beta \\ -\beta/\alpha \\ 0 \end{bmatrix} = \left(\beta - \alpha\frac{\beta}{\alpha}\right)\boldsymbol{c_1} + 0\boldsymbol{c_2}$$

- What else can we cancel out?
- An infinite number of non-zero vectors **x** can map to a zero-vector **y**
- Called the **right null-space** of A.

Rank

- Rank of a nxn matrix **A** – number of linearly independent columns (**or rows**) of A / the dimension of the span of the columns
- Matrices with *full rank* (n x n, rank n) behave nicely: can be inverted, span the full output space, are one-to-one.
- Matrices with *full rank* are machines where every knob is useful and every output state can be made by the machine

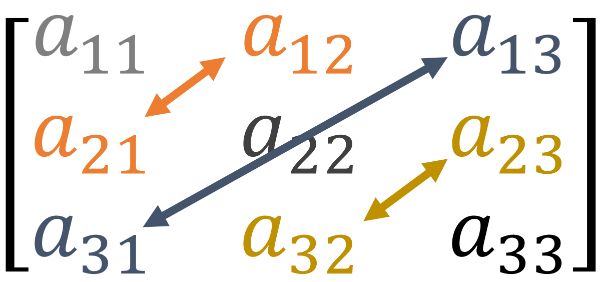
Matrix Inverses

- Given $\boldsymbol{y} = \boldsymbol{Ax}$, **y** is a linear combination of columns of **A** proportional to **x**. If A is full-rank, we should be able to invert this mapping.
- Given some **y** (output) and **A**, what **x** (inputs) produced it?
- **x** = **A**$^{-1}$**y**
- Note: if you don't need to compute it, **never ever compute it.** Solving for **x** is much faster and stable than obtaining **A**$^{-1}$.

Bad: `y = np.linalg.inv(A).dot(y)`
Good: `y = np.linalg.solve(A, y)`

Symmetric Matrices

- Symmetric: $\boldsymbol{A}^T = \boldsymbol{A}$ or $\boldsymbol{A}_{ij} = \boldsymbol{A}_{ji}$
- Have **lots** of special properties



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Any matrix of the form $\boldsymbol{A} = \boldsymbol{X}^T\boldsymbol{X}$ is symmetric.

Quick check:

$$\boldsymbol{A}^T = \left(\boldsymbol{X}^T\boldsymbol{X}\right)^T$$
$$\boldsymbol{A}^T = \boldsymbol{X}^T\left(\boldsymbol{X}^T\right)^T$$
$$\boldsymbol{A}^T = \boldsymbol{X}^T\boldsymbol{X}$$

Special Matrices: Rotations

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

- Rotation matrices $\boldsymbol{R}$ rotate vectors and ***do not change vector L2 norms*** ($\|\boldsymbol{Rx}\|_2 = \|\boldsymbol{x}\|_2$)
- Every row/column is unit norm
- Every row is linearly independent
- Transpose is inverse $\boldsymbol{RR}^T = \boldsymbol{R}^T\boldsymbol{R} = \boldsymbol{I}$
- Determinant is 1 (otherwise it's also a coordinate flip/reflection), eigenvalues are 1

Next Time:
More Linear Algebra
+ Image Filtering