Lecture

What is Vectorization

$z = w^T x + b$ $\quad w = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$ $\quad x = \begin{bmatrix} \vdots \\ \vdots \end{bmatrix}$

$w \in \mathbb{R}^{n_x}$ $\quad x \in \mathbb{R}^{n_x}$

Non vectorized:

```
z=0
for i in Range(n_x):
    z += w[i] * x[i]
z += b
```

↳ Really Slow. Doing the whole vector multiplication explicitly.

Vectorized:

In Python or Numpy

```
z = np.dot(w, x) + b
```

↳ Faster.

GPU
CPU
} SIMD-Style instruction multiple data.

Vectorization allows you to take advantage of the parallelism in hardware, in both CPU and GPU.

But for GPU we really go all out for SIMD calculations

∴ Vectorize when possible

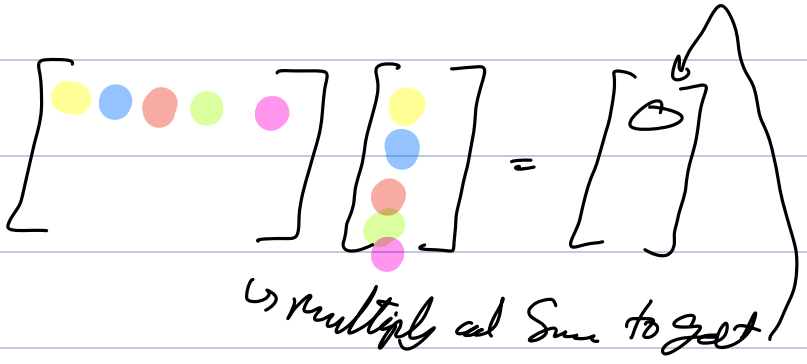
Lec 11 More vectorization

Neural Network Programming:

Avoid Explicit for-loops → to be faster.

$$\vec{u} = A\vec{v}$$

A: matrix



↳ multiply and sum to get

$$\vec{u}_i = \sum_j A_{ij} \vec{v}_j$$

u = np.zeros((n, 1))

for i ...

  for j ...

    u[i] += A[i][j] * u[j]

} two for loops.

u = np.dot(A, v) ⇐ vectorized version

eg. $v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$ in many cases want to apply e^{v_i} for all i.

$$\Rightarrow u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

Non-vectorized

u = np.zeros((n, 1))

for i in range(n):

  u[i] = math.exp(v[i])

Vectorized

input may as np.

u = np.exp(v)

// np.log(v)

// np.abs(v)

Numpy BUILTIN Fxn ⇒ // np.maximum(v, 0)

// v**2 → Square

// 1/v → Inverse.

$J = 0,\ dw1 = 0,\ dw2 = 0,\ db = 0$

for i = 1 to m: → 1 for loop

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$ } 2nd for loop, $n_x = 2$. if n_x was large $dw_1 \ldots dw_n$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J = J/m,\ dw_1 = dw_1/m,\ dw_2 = dw_2/m,\ db = db/m$

get rid of this for loop.

$J = 0,\ dw1 = 0,\ dw2 = 0,\ db = 0$ $dw = np.zeros((n_x, 1))$

for i = 1 to m:

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

$J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

get rid of for loop → $dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$ $dw \mathrel{+}= x^{(i)} dz^{(i)}$

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$

$db \mathrel{+}= dz^{(i)}$

$J = J/m,\ dw_1 = dw_1/m,\ dw_2 = dw_2/m,\ db = db/m$

dw /= m

Lec 12

VECTORIZED LOGISTIC REGRESSION

Forward Prop Step

$$z^{(1)} = w^T x^{(1)} + b \qquad z^{(2)} = w^T x^{(2)} + b$$
$$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)})$$

--- m training examples

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

(n_x rows, m columns)

(n_x, m)

$\langle n_x, m \rangle$

$\mathbb{R}^{n_x \times m}$

$$[z^{(1)} \; z^{(2)} \; \cdots \; z^{(m)}] = w^T X + [b \; \cdots \; b]$$

w^T: $1 \times n_x$, X: $n_x \times m$ $\Rightarrow$ $1 \times m$

$[b \; \cdots \; b]$: $1 \times m$ matrix or m-dim row vector

$w^T = 1 \times [w_1 \; w_2 \; \ldots \; w_m]$

$[z^{(1)} \; z^{(2)} \; \ldots \; z^{(m)}] = [\underbrace{w^T x^{(1)} + b}_{z^{(1)}}, \underbrace{w^T x^{(2)} + b}_{z^{(2)}} \ldots w^T x^{(m)} + b]$

$Z = [z^{(1)} \; z^{(2)} \; \ldots \; z^{(m)}]$

$(1, m)$

$Z = np.dot(w.T, x) + b$ → Real # (1,1) → will expand it out or BROADCAST to $(1, m)$

$A = [a^{(1)} a^{(2)} \ldots a^{(m)}] = \sigma(Z)$

No longer need to loop over m training examples

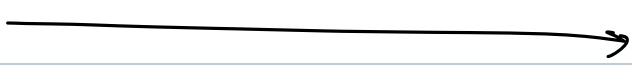
Lec 13 - Vectorizing Logistic Regression's Output

Vectorized Computation

for example i $dz^{(1)} = a^{(1)} - y^{(1)}$ $dz^{(2)} = a^{(2)} - y^{(2)}$

$\ldots dz^{(m)} = a^{(m)} - y^{(m)}$.

So $\Rightarrow$ $dZ = [dz^{(1)}, dz^{(2)} \ldots dz^{(m)}]$



a m-dim row vector

$A = [a^{(1)} \dots a^{(m)}]$

$Y = [y^{(1)} \dots y^{(m)}]$

$dZ = A - Y = [\underbrace{a^{(1)} - y^{(1)}}_{dz^{(1)}} \quad \underbrace{a^{(2)} - y^{(2)}}_{dz^{(2)}}]$

$dw = 0$

$dw \mathrel{+}= x^{(1)} dz^{(1)}$

$dw \mathrel{+}= x^{(2)} dz^{(2)}$

$\vdots$ m

$dw \mathrel{/}= m$

db

$db \mathrel{+}= dz^{(1)}$

$db \mathrel{+}= dz^{(2)}$

$\vdots$

$db = dz^{(m)}$

$db \mathrel{/}= m$

Vectorize.

Summing up all the dz

$dw = \frac{1}{m} X dz^T$

$X = [x^{(1)} \dots x^{(m)}]$ ↓ n

$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$

$db = \frac{1}{m} db.\text{Sum}(dz)$

↳ Reduce op

$dz^T = \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}_{m \times 1}$ $\quad X\, dz^T \Rightarrow n \times 1 = [\,]$

$n \times m \quad m \times 1$

Implementing Logistic Regression

INEFFICIENT w/ LOOPING

```
J = 0, dw1 = 0, dw2 = 0, db = 0
for i = 1 to m:
    z(i) = w^T x(i) + b
    a(i) = σ(z(i))
    J += -[y(i) log a(i) + (1 - y(i)) log(1 - a(i))]
    dz(i) = a(i) - y(i)
    dw1 += x1(i) dz(i)
    dw2 += x2(i) dz(i)
    db += dz(i)
J = J/m, dw1 = dw1/m, dw2 = dw2/m
db = db/m
```

$dw \mathrel{+}= x^{(i)} dz^{(i)}$

1) $Z = w^T X + b = np.dot(w.T, X) + b$ for all i

2) $A = \sigma(Z)$

3) $dZ = A - Y$

4) $dw = \frac{1}{m} X\, dZ^T$

5) $db = \frac{1}{m} np.sum(dZ)$

Single iteration of Gradient descent. Will need to loop for num iterations

$w := w - \alpha\, dw$

$b := b - \alpha\, db$

Lec 14 • Broadcasting in Python

Broadcasting Example

Calories from Carbs, Proteins, Fats in 1 oz of diff foods

	Apple	Beef	Eggs	Potatoes
Carbs	5.6	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.1

$= A_{(3,4)}$

= 59 cal ⇒ % cal from Carb = $\frac{5.6}{59} \approx 94.9\%$

Calculate % of calories from Carbs, Protein, Fat.

Sum up the cols to get total calories, then divide the first row entries to get % of cal from Carbs, Proteins and Fat.

$A = (3,4)$ ↓ # rows, # of col.

Sum axis 0, to go along the first axis of 3.

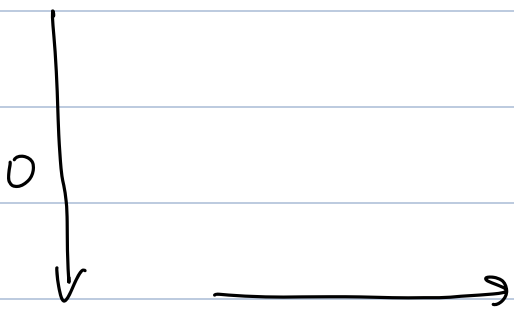
Sum along axis 0 → to go into each row and sum?

```python
import numpy as np

A = np.array([[56.0, 0.0, 4.4, 68.0],
              [1.2, 104.0, 52.0, 8.0],
              [1.8, 135.0, 99.0, 0.9]])

print(A)


cal = A.sum(axis=0) #sum vertically
print(cal)
# [ 59.  239.  155.4  76.9]

#dividing the 3x4 matrix A by the 1x4 matrix cal through broadcasting
percentage = 100*A/cal.reshape(1,4) #broadcasting
print(percentage)

# [[94.91525424  0.          2.83140283 88.42652796]
#  [ 2.03389831 43.51464435 33.46203346 10.40312094]
#  [ 3.05084746 56.48535565 63.70656371  1.17035111]]
```

Axis 0

↳ Sum

vertically

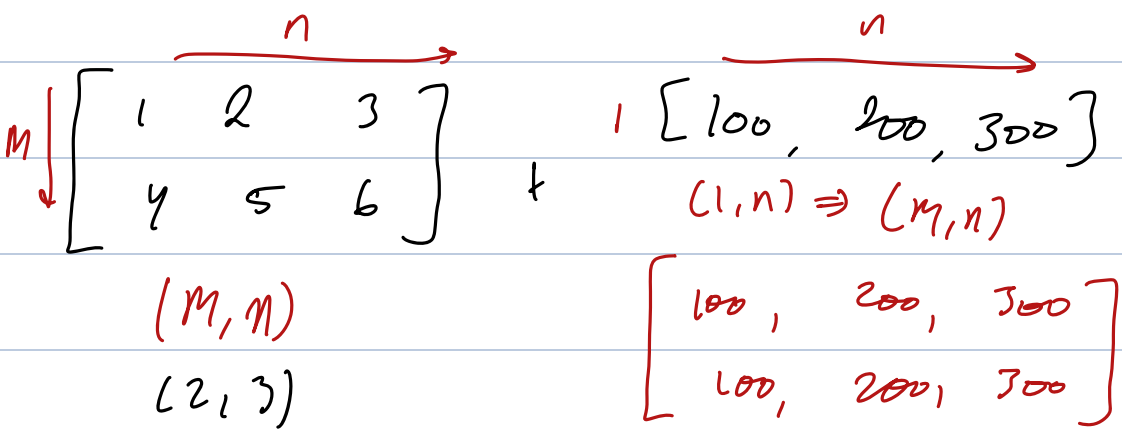


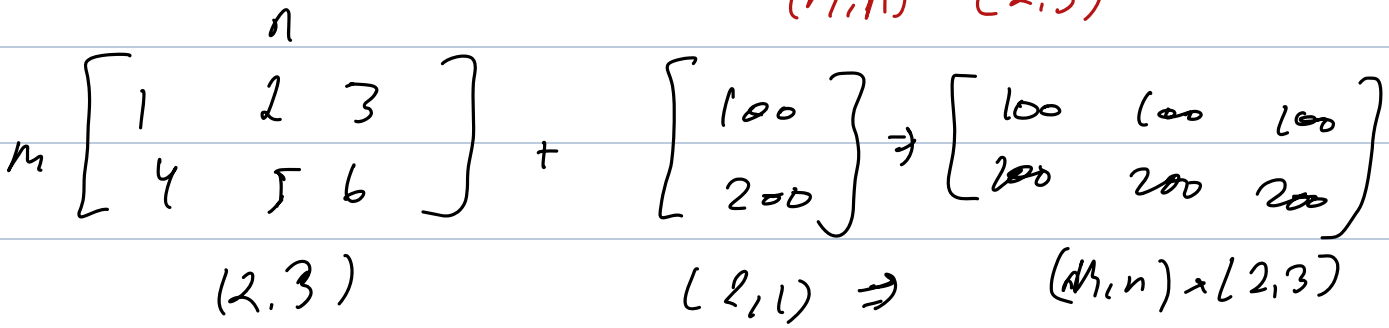
A - (3,4)

Cal = (1,4)

Broadcasting

$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \Rightarrow \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}$$





General Idea

(m,n) matrix

$\frac{+}{-}$ / $\times$ /

(1,n) ⇒ (m,n) , copy m times

(m,1) ⇒ (m,n) copy n times

apply elementwise

(m,1)

m [:] 1

\+ IR 100 ⇒ [100, 100, ⋮, 100] (m,1)

(1,n)

[...] + IR 100 ⇒ [100 ...] (1 x m)

Lec 15 → Python / Numpy Vectors

```python
import numpy as np


a = np.random.randn(5) #random 1-d tensor (array) of 5 elements
print(a)
print(a.shape) # (5,)

print(a.T) # [ 0.  0.  0.  0.  0.] -> no change

print(np.dot(a, a.T)) # 5.0 -> dot product of a and a.T -> scalar (not as expected)

a = np.random.randn(5,1) #random 2-d tensor (array) of 5x1 elements
print(a)
print(a.shape) # (5,1) [5 rows, 1 column] [[0], [0], [0], [0], [0]]
print(a.T) # [[ 0.  0.  0.  0.  0.]] -> transpose of a -> 1x5
print(np.dot(a, a.T)) # [[ 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.]] -> 5x5 matrix

#assert(a.shape == (5,1)) #assertion error if not true


```

Rank 1 array → don't use.

↳ this by $a = a.\text{reshape}(5,1)$

5x1 matrix

↳ col vector ✓ or 1.5 → row vector

✓ to make sure.