Vectorization

→ art of getting rid of explicit for loops in the code

$$z = w^T x + b$$

$$w = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \quad x = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix} \qquad w \in R^{n_x} \quad x \in R^{n_x}$$

without vectorized by for loop

For Non vectorized

```
z = 0
for i in range (n - x):
    z += w[i] * x[i]
z += b
```

much faster

Vectorized

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

$w^T x$ — command in numpy

Example code: import numpy as np

```
a = np.array([1,2,3,4])
print a
```

O/p: [1 2 3 4]

We use that to time how long diff. operations take

```
import time

a = np.random.rand(1000000)
b = np.random.rand(1000000)
tic = time.time()
c = np.dot(a,b)
toc = time.time()
print("Vectorized version" + str(1000*(toc-tic)) + "ms")
```

Create multidimensional array with random values

Measure current time

Current time (after − before) operation

× 1000 — so we can express in millisecond

O/p about 1.5 milliseconds

```
c = 0
tic = time.time()
for i in range(1000000):
    c += a[i] * b[i]
toc = time.time()

print("For loop: " + str(1000*(toc-tic)) + "ms")
```

O/p: 475 ms (took much more time to compute c)

Code runs 300 times faster by vectorization

→ A lot of scaleable deep learning implementations are done on a GPU or a graphics processing unit.

→ Both CPU & GPU have parallelization implementations are done on GPU or instructions

→ They are sometimes called SIMD instructions which stands for single instruction multiple data. → what this basically means if we use built in func. such as np. function or other func. that dont require implementing for loop.

→ Enables python pi to take much better advantage of parallelism to do your computation much faster → This is true for both GPU & CPU but GPU is better

→ Whenever possible avoid using explicit for loops

More vectorization examples

Eg $u = Av$

vector u is product of matrix A & vector v

$$u_i = \sum_j A_{ij} v_j$$

non vec. implementation

```
u = np.zeros((n,1))
for i ... {
    for j ... {
        u[i] += A[i][j] * v[j]
```

for vectorized version it would be

$u = np.dot(A, v)$

Eg:- Lets say we need to apply exponential operation on every element of matrix / vector

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \quad u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

non vectorized implementation:

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(v[i])
```

→ Python & NumPy have many built in func that allow u to compute these vectors with just a single call to a single function

by vectorization

```
import numpy as np
u = np.exp(v)
```

(u = o/p vector, v = i/p vector)

Other useful func. in numpy

np.log(v) — log value
np.abs(v) — absolute value
np.maximum(v) — computes element wise maximum to take maximum of every element of v with 0
v**2 — element wise square of v
1/v — element wise inverse of v

Logistic regression derivatives

J = 0, dw1 = 0, dw2 = 0, db = 0
for i = 1 to m:
$z^{(i)} = w^T x^{(i)} + b$
$a^{(i)} = \sigma(z^{(i)})$
$J += -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$
$dz^{(i)} = a^{(i)} - y^{(i)}$
$dw_1 += x_1^{(i)} dz^{(i)}$
$dw_2 += x_2^{(i)} dz^{(i)}$
$db += dz^{(i)}$

(here nx = 2 but if more features we will need a for loop)

J = J/m, dw1 = dw1/m
dw2 = dw2/m, db = db/m

we need to get rid of this second for loop by making dw a vector

numpy.zeros() – create an array of desired shape & size with each element as 0

PAGE NO.
DATE / /

dw = np.zeros((n_x, 1))

Instead of
$dw_1 += x_1^{(i)} dz^{(i)}$
$dw_2 = x_2^{(i)} dz^{(i)}$ etc

we use
$dw += x^{(i)} dz^{(i)}$

Instead of
$dw_1 = dw_1/m$
$dw_2 = dw_2/m$ etc

we have
$dw /= m$

Vectorizing logistic regression:

⇒ We can vectorize the implementation of logistic regression so they can process an entire training set, ie implement a single elevation wrt an entire training set without even a single for loop

forward prop

⇒ $z^{(1)} = w^T x^{(1)} + b$ $z^{(2)} = w^T x^{(2)} + b$
$a^{(1)} = \sigma(z^{(1)})$ $a^{(2)} = \sigma(z^{(2)})$
.... m times

⇒ We can do this without for loop

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

n_x by m dimensional matrix

1 x m matrix

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

w^T: $1 \times n_x$ (as w: $n_x \times 1$); X: $n_x \times m$; $[b \; b \; b \; \dots]$: $1 \times m$ matrix

$$= [w^T x^{(1)} \quad w^T x^{(2)} \quad \dots \quad w^T x^{(m)}] + [b \quad b \quad b \quad \dots \quad b]_{1 \times m}$$

$$= [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \dots \quad w^T x^{(m)} + b]_{1 \times m}$$

$$= [z^{(1)} \quad z^{(2)} \quad \dots \quad z^{(m)}]$$

$$= Z$$

$$Z = [z^{(1)} \quad z^{(2)} \quad \dots \quad z^{(m)}]$$

numpy command for this

```
Z = np.dot(w.T, X) + b
```

b is a normal no (1,1)

This is called broadcasting in Python

but when we do the operation it is automatically converted to a 1xm matrix of bs

→ Now we need to find

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

Vectorizing Logistic Regression's gradient computation

→ We can use vectorization to perform gradient computation for all M training samples

Since

$dz^{(1)} = a^{(1)} - y^{(1)}$

$dz^{(2)} = a^{(2)} - y^{(2)}$ ... m times

$dZ = [dz^{(1)}\ dz^{(2)} \dots dz^{(m)}]$

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

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

$dZ = A - Y = [a^{(1)} - y^{(1)}\ a^{(2)} - y^{(2)}, \dots]$

Also we have

$dw = 0$

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

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

⋮ m times

$dw /= m$

and

$db = 0$

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

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

⋮

$db += dz^{(m)}$

$db /= m$

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

$$= \frac{1}{m} np.sum(dz)$$

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

$$= \frac{1}{m} \begin{bmatrix} x^{(1)} & \cdots & x^{(m)} \end{bmatrix}_{n \times m} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}_{m \times 1}$$

$$= \frac{1}{m} \underbrace{\left[x^{(1)} dz^{(1)} + \cdots + x^{(m)} dz^{(m)} \right]}_{n \times m \text{ times}}_{n \times 1}$$

$\therefore$

$$z = w^T x + b$$
$$= np.dot(w.T, X) + b$$
$$A = \sigma(z)$$
$$dz = A - Y$$
$$dw = \frac{1}{m} X dz^T$$
$$db = \frac{1}{m} np.sum(dz)$$

With this we have implemented single iteration of gradient descent

$$w = w - \alpha\, dw$$
$$b = b - \alpha\, db$$

This would be the gradient descent update.

But still need to loop over the number of iteration

→ So if we need 1000 iterations of gradient descent we must loop 1000 times

→ This is the outermost loop which we can't get rid of

Broadcasting in python

$$\begin{bmatrix} x_1 & y_1 & z_1 & P_1 \\ x_2 & y_2 & z_2 & P_2 \\ x_3 & y_3 & z_3 & P_3 \end{bmatrix}$$

we want to sum up x, y, z & P and find how much % of x is x_1, x_2, x_3 & same for y. & z & P also

→ But we want to do this without an explicit for loop

code:

```
import numpy as np
A = np.array([[x1, y1, z1, P1],
              [x2, y2, z2, P2],
              [x3, y3, z3, P3]])

cal = A.sum(axis = 0)
```

(to sum horizontally use 1) axis = 0 ~~means~~ sum vertically

(cal will be [Σx Σy Σz ΣP])

```
percentage = 100* A / cal.reshape(1,4)
print(percentage)
```

we didnt need this here since already 1x4 ∴ redundant

Eg $\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100$ → Python would convert this to $\begin{bmatrix}100\\100\\100\\100\end{bmatrix}$

∴ O/P would be: $\begin{bmatrix}101\\102\\103\\104\end{bmatrix}$

Eg $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100 & 200 & 300\end{bmatrix}$

Python would convert this to $\begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix}$

O/P would be: $\begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$

Eg $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + \begin{bmatrix}100\\200\end{bmatrix}$

Python will convert this to

$\begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix}$

General principle of broadcasting
- If u have an (m, n) matrix & we +/-/*/÷ by a (1, n) matrix, then this will copy m times into a (m x n) matrix then apply

addition, sub., multiplic, division similarly for an (m,1) matrix

→ it will be copied n times if we +/−/*/÷ by a real no it is copied in each element of the matrix having same size & shape

Notes on Python – Numpy

Eg import numpy as np
a = np.random.randn(5)
Print(a)

O/P: [v_1 v_2 v_3 v_4 v_5]
↓
5 random variables

Print(shape) print(a.shape)

O/P: (5,) ← This is called a rank 1 array in python and is neither row nor column vector

Print(a.T) ← a transpose

Print(np.dot(a, a.T))

O/P: U get a number

→ It is recommended to not use data structures where shape is a rank 1 array

Instead take a to be
a = np.random.randn(5,1)
print a
o/p: a will be a 5x1 column vector

→ If u are not sure of your shape use an assertion statement
Eg assert(a.shape == (5,1))

→ If u end up with a rank 1 array use .reshape
Eg a=a.reshape((5,1))

Note:
→ If -ve is added to main / test set of a it is bcoz we are going to perform sqrt on them

→ Trick when u want to convert matrix X (a,b,c,d) to X_flatten (b*c*d, a)
use
X_flatten = X.reshape(X.shape[0], -1)

refers to unknown dimension that reshape() func. calculates for you