Vectorization.

what is it?

In logistic regression we need to calculate.

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

DOMS

$$w^T = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

__/__/__

we can calculate $w^T x$ using a for loop
↳ bad practice. for i in range(n-x)
$z += w[i] * x[i]$

☆ Vectorized method.

$$z = \underbrace{np.dot(w, x)}_{\downarrow} + b$$

Much faster.

Taking an example of np.dot()
✓ of 2 arrays with 10^6 elements

done live

vectorized version takes ~ 1.5 ms

non-vectorized takes ~ 475 ms huge difference

vectorised is ~300 times faster !!!

why tho? why is it faster?

→ CPU's and GPU's have some parallelization features
SIMD ↓ single instruction multiple data
GPU's ↓ have parallelization turned on by default
parallelization is what makes it faster

np internally executes it using parallization features

normal forloop doesn't

Applying Vectorization to Logistic Regression

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

. . . . up til n

✱ Forward propogation

we represent input as $n \times m$ dimensional matrix

(n = no. of inputs/features in 1 example; m = no. of training examples)

$$X_{n \times m} = \begin{bmatrix} | & | & | & & | \\ x_1 & x_2 & x_3 & \cdots & x_m \\ | & | & | & & | \end{bmatrix} \qquad w = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$$

To calculate z without using for loop

$$\boxed{Z = w^T X}$$

$$Z = [z_1 \; z_2 \; z_3 \cdots z_m] = w^T X + [b \; b \cdots b]$$

(m times)

$$w^T \rightarrow [\theta_1 \; \theta_2 \cdots \theta_n] \begin{bmatrix} | & | & | & & | \\ x_1 & x_2 & x_3 & \cdots & x_m \\ | & | & | & & | \end{bmatrix} \leftarrow X$$

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

numpy handles converting b into arrays and making z a resultant vector.

now $A = \sigma(Z)$

(A = another array, Z = array)

✱ Backward Propogation.

operations: $dz^{(1)} = a^{(1)} - y^{(1)} \quad \cdots \quad dz^{(m)} = a^{(m)} - y^{(m)}$

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

$$A = [a^{(1)} \cdots a^{(m)}] \qquad Y = [y^{(1)} \cdots y^{(m)}]$$

In vectorized method, there's no j bcz every thing is represented as arrays.

$A^{(i)}$ → i^{th} training set

$A_{(j)}$ → j^{th} feature in i^{th} training set

Algo :-

→ $dw = 0$

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

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

⋮

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

$dw /= m$

↓ vectorized method

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

↓

$$= \frac{1}{m} \begin{bmatrix} \vdots & \vdots & & \vdots \\ x_1 & x_2 & \cdots & x_m \\ \vdots & \vdots & & \vdots \end{bmatrix} \begin{bmatrix} dz^1 \\ dz^2 \\ dz^3 \\ \vdots \\ dz^m \end{bmatrix}$$

$$dw = \frac{1}{m} \left[x_1 dz^1 + \cdots + x^m dz^m \right]$$

$db = 0$

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

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

$db^{(3)} += dz^{(3)}$

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

$db /= m$

↓

vectorized method

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

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

Final code.

forward propagation:

$Z = w^T x + b$

$= \text{np.dot}(w.T, X) + b$

$A = \sigma(Z)$

backward propagation:

$dZ = A - Y$

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

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

Then we update dw, db for Gradient descent

↓

So in the end we do need for loops for Gradient descent

↑

finish step 1 of Gra Des

↑

do Grad Desc Again

↑

requires for loop

DOMS