Vectorization

What is vectorization?

In logistic regression, you need to compute $z = w^T x + b$ where w and x are vectors (may be some large vector)

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

To compute $z = w^T x$ **:**

- Non-vectorized method:

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

  -> very slow

- Vectorized method:

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

  -> Much faster

Vectorization Code Demo:

```
[2]: import numpy as np

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

     [1 2 3 4]

[3]: 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")

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

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

     Vectorized version: 0.9770393371582031ms
     For loop version: 396.4099884033203ms
```

Neural network programming guideline

Whenever possible, avoid explicit for-loops.

Say you need to apply the exponential operation on every element of a matrix/vector.

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

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

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

Python numpy provides many different element-wise operators:

```
np.log(v)
np.abs(v)
np.maximum(v, 0)
v**2        1/v
```

Logistic regression derivatives

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

for i = 1 to m:

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

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

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

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

for $j = 1 \ldots n_x$, $dw_j \mathrel{+}= \ldots$

$$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$

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

$n_x = 2$

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

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

Instead of using explicit for loop like the above example, we can do the following way:

$J = 0,\ \boxed{\cancel{dw1 = 0,\ dw2 = 0}},\ db = 0$ $\quad dw = np.zeros((n\_x, 1))$

for i = 1 to m:

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

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

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

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

for $j = 1 \ldots n_x$, $dw_j \mathrel{+}= \ldots$

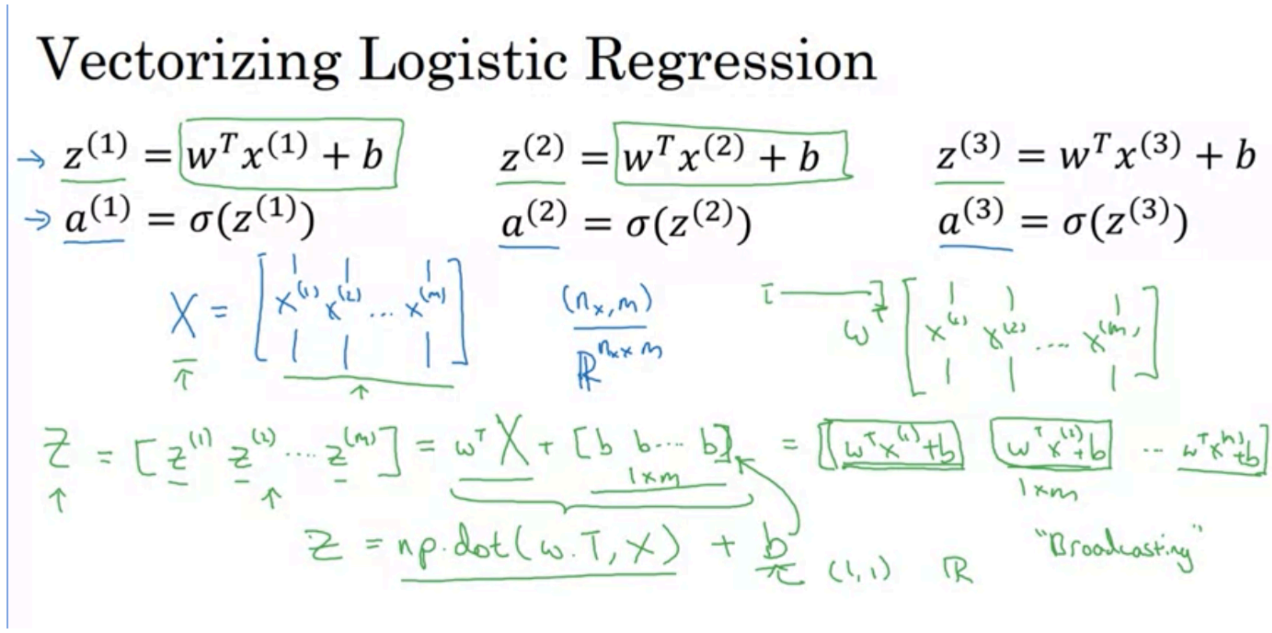
$$\boxed{\cancel{dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}} \\ \cancel{dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}}}$$

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

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

$J = J/m,\ \boxed{\cancel{dw_1 = dw_1/m,\ dw_2 = dw_2/m}},\ db = db/m$

$dw /= m.$



Z is going to be a 1xm matrix that contain all of the lowercase z's

To recap, what we've seen on this slide is that instead of needing to loop over M training examples to compute lowercase Z and lowercase A, one of the time, you can implement this one line of code, to compute all these Z's at the same time.

So this is how you implement a vectorize implementation of the four propagation for all M training examples at the same time.