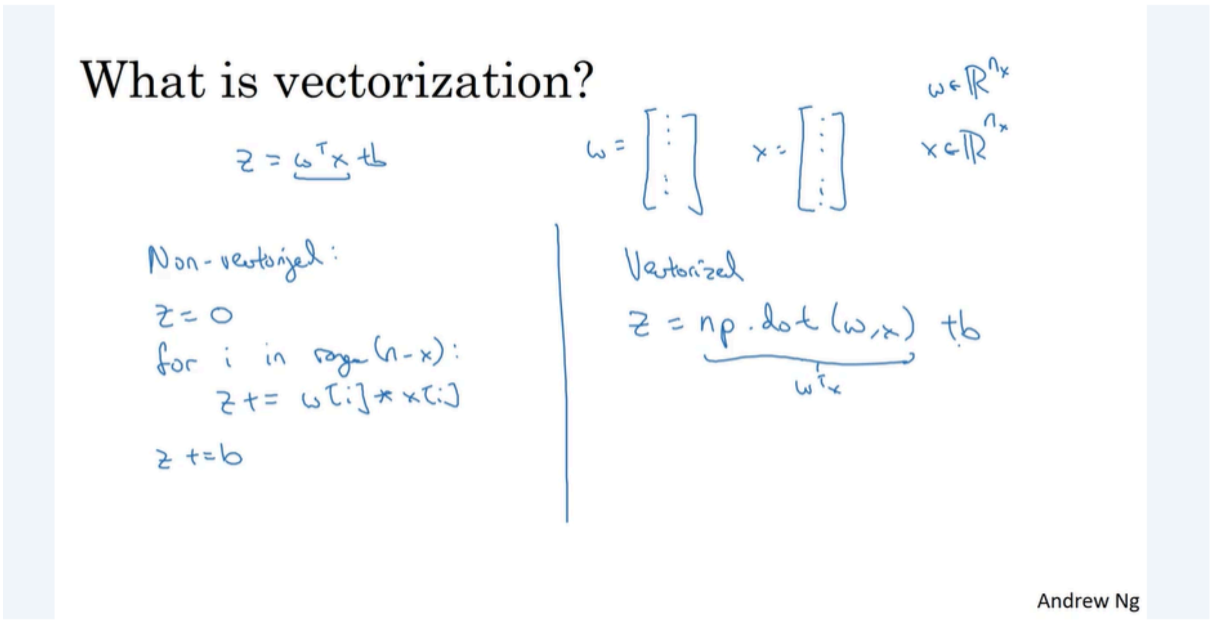
Python and Vectorization

Vectorization



What is vectorization?

$z = \underbrace{w^T x} + b$

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

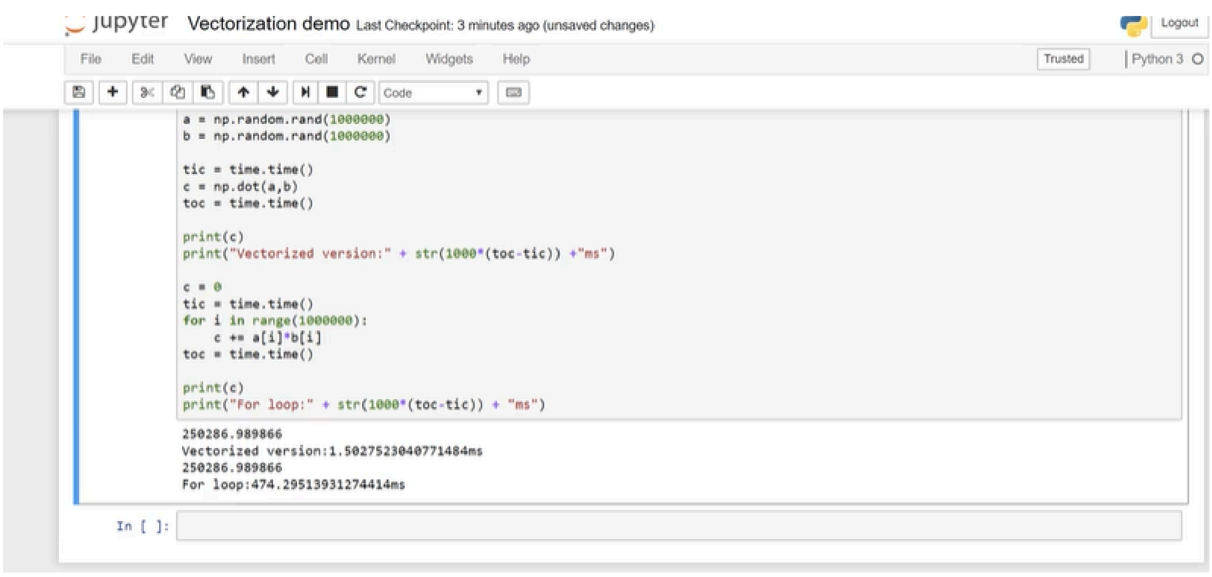
Non-vectorized:

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

Vectorized

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

Andrew Ng



jupyter Vectorization demo Last Checkpoint: 3 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Code

```python
a = np.random.rand(1000000)
b = np.random.rand(1000000)

tic = time.time()
c = np.dot(a,b)
toc = time.time()

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

```
250286.989866
Vectorized version:1.5027523040771484ms
250286.989866
For loop:474.29513931274414ms
```

In [ ]:

The vectorize version took 1.5 milliseconds. The explicit for loop and non-vectorize version took about 400, almost 500 milliseconds. The non-vectorize version took something like 300 times longer than the vectorize version. With this example you see that if only you remember to vectorize your code, your code actually runs over 300 times faster. Let's just run it again. Just run it again. Yeah. Vectorize version 1.5 milliseconds seconds and the for loop.

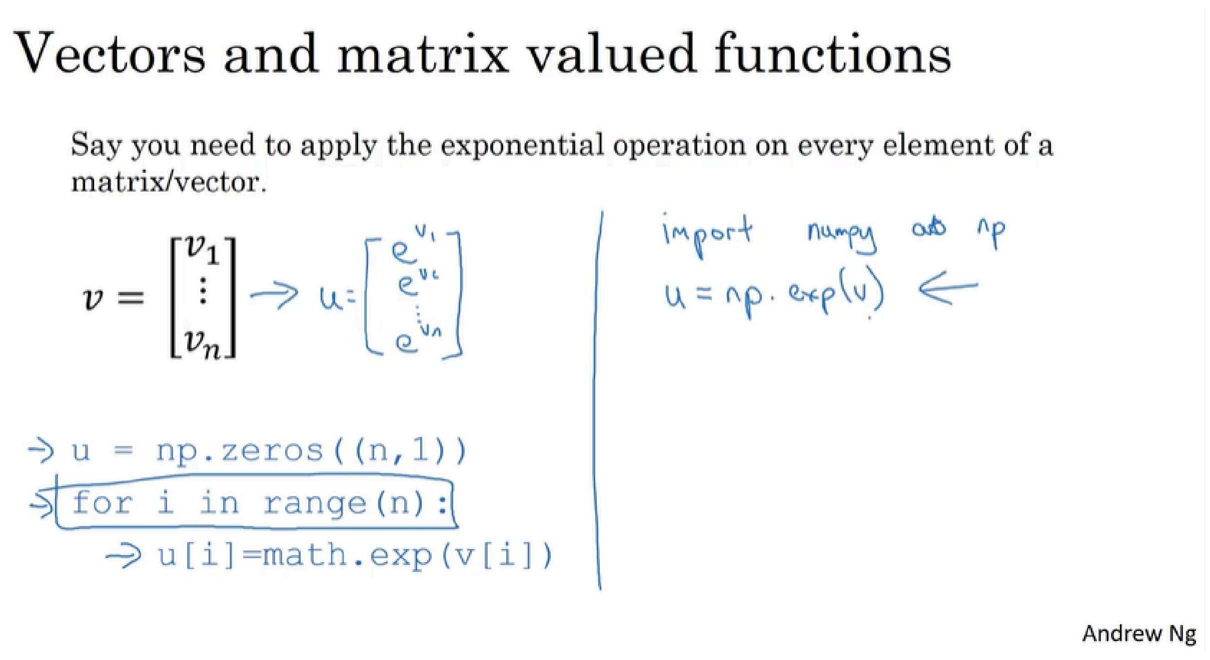
So 481 milliseconds, again, about 300 times slower to do the explicit for loop. If the engine x slows down, it's the difference between your code taking maybe one minute to run versus taking say five hours to run.

And when you are implementing deep learning algorithms, you can really get a result back faster. It will be much faster if you vectorize your code. Some of you might have heard that a lot of scaleable deep learning implementations are done on a GPU or a (graphics processing unit). But all the demos I did just now in the Jupiter notebook where actually on the CPU. And it turns out that both GPU and CPU have parallelization instructions.

They're sometimes called SIMD instructions. This stands for a single instruction multiple data. But what this basically means is that, if you use built-in functions such as this np.function or other functions that don't require you explicitly implementing a for loop.

> The rule of thumb to remember is whenever possible, avoid using explicit for loops.

More Vectorization Examples



Vectors and matrix valued functions

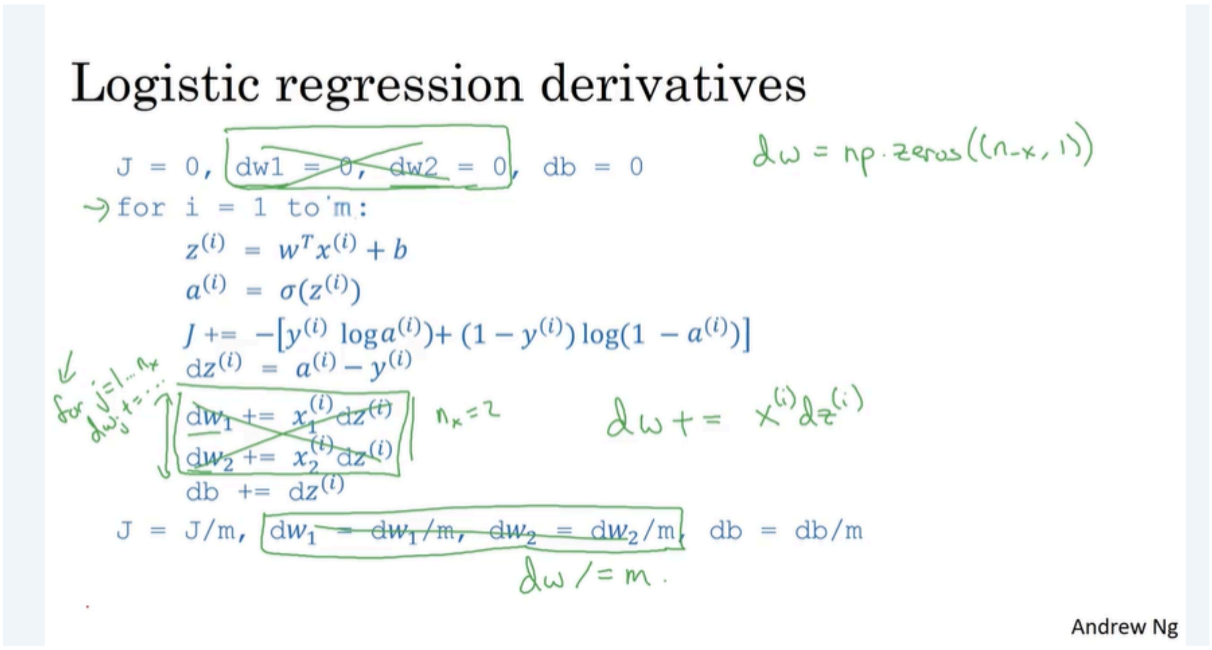
Say you need to apply the exponential operation on every element of a matrix/vector.

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

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

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

Andrew Ng

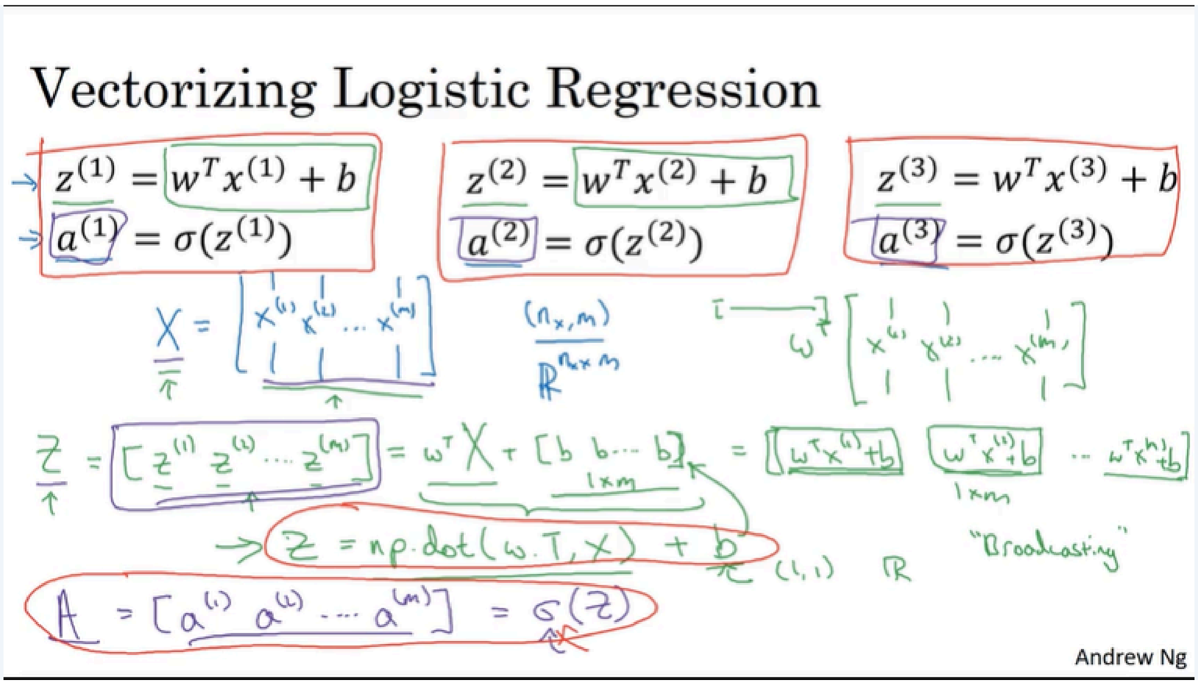


Logistic regression derivatives

```
J = 0, ~~dw1 = 0, dw2 = 0~~, db = 0          $dw = np.zeros((n_x, 1))$
→ for i = 1 to m:
      $z^{(i)} = w^T x^{(i)} + b$
      $a^{(i)} = \sigma(z^{(i)})$
      $J \mathrel{+}= -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$
      $dz^{(i)} = a^{(i)} - y^{(i)}$
      ~~$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$~~      $n_x = 2$      $dw \mathrel{+}= x^{(i)} dz^{(i)}$
      ~~$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$~~
      for j = 1 … n_x
          $dw_j \mathrel{+}= …$
      db += $dz^{(i)}$
J = J/m, ~~$dw_1 = dw_1/m$, $dw_2 = dw_2/m$~~, db = db/m
                 $dw \mathrel{/}= m$.
```

Andrew Ng

Vectorizing Logistic Regression



Vectorizing Logistic Regression

$$\boxed{z^{(1)} = w^T x^{(1)} + b \quad a^{(1)} = \sigma(z^{(1)})} \qquad \boxed{z^{(2)} = w^T x^{(2)} + b \quad a^{(2)} = \sigma(z^{(2)})} \qquad \boxed{z^{(3)} = w^T x^{(3)} + b \quad a^{(3)} = \sigma(z^{(3)})}$$

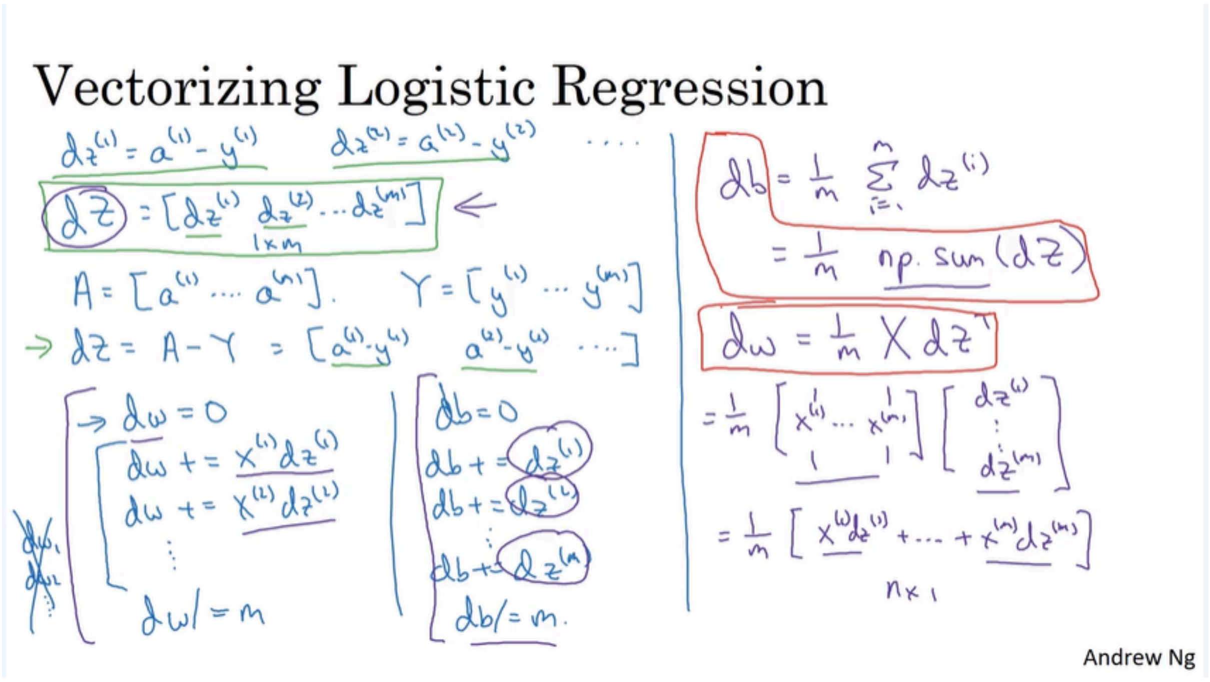
$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \qquad \frac{(n_x, m)}{\mathbb{R}^{n_x \times m}} \qquad w^T \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

$$Z = [z^{(1)} \; z^{(2)} \cdots z^{(m)}] = \underbrace{w^T X + \underbrace{[b \; b \cdots b]}_{1 \times m}} = \underbrace{[w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad \cdots \quad w^T x^{(m)} + b]}_{1 \times m}$$

$$\rightarrow Z = np.dot(w.T, X) + b \qquad (1,1) \quad \mathbb{R} \qquad \text{"Broadcasting"}$$

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

Andrew Ng



Vectorizing Logistic Regression

$dz^{(1)} = a^{(1)} - y^{(1)}$ $\quad dz^{(2)} = a^{(2)} - y^{(2)}$ $\quad \dots$

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

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

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

```
→ dw = 0
  dw += $x^{(1)} dz^{(1)}$
  dw += $x^{(2)} dz^{(2)}$
  ⋮
  dw /= m
```

~~dw_1~~, ~~dw_2~~

```
db = 0
db += $dz^{(1)}$
db += $dz^{(2)}$
⋮
db += $dz^{(m)}$
db /= m.
```

$$\boxed{db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)} = \frac{1}{m} \text{np.sum}(dZ)}$$

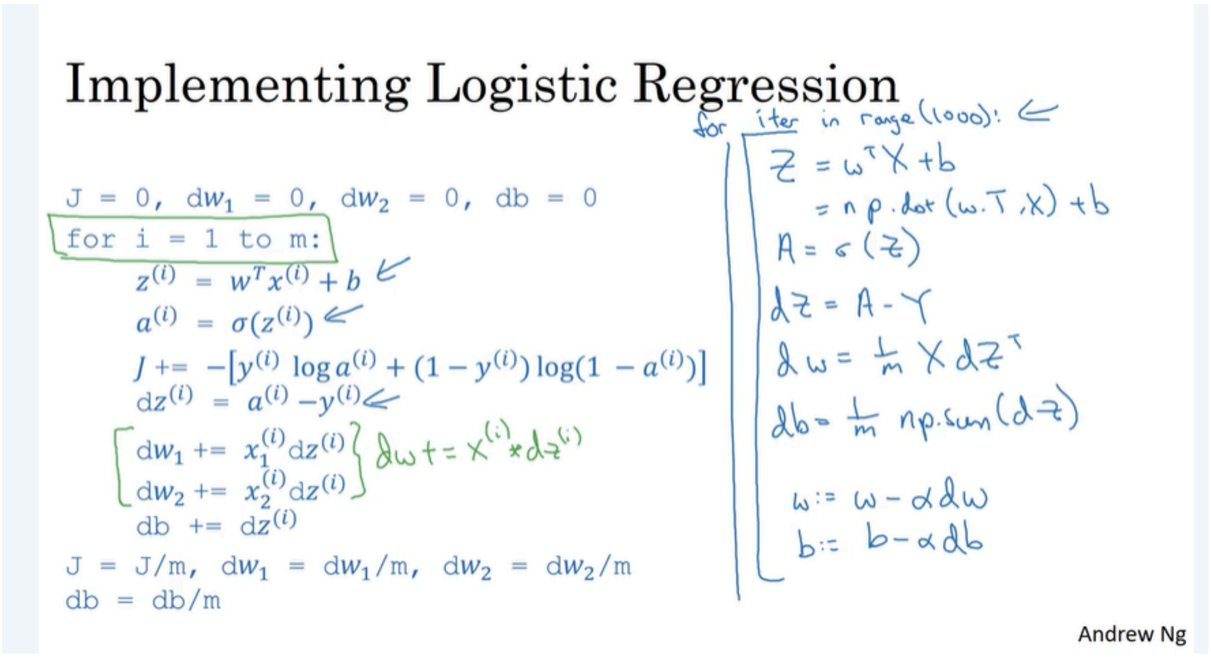
$$\boxed{dw = \frac{1}{m} X dZ^T}$$

$$= \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

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

Andrew Ng

The left side is non-vectorize approach, and the right side is the vectorize approach.



Implementing Logistic Regression

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

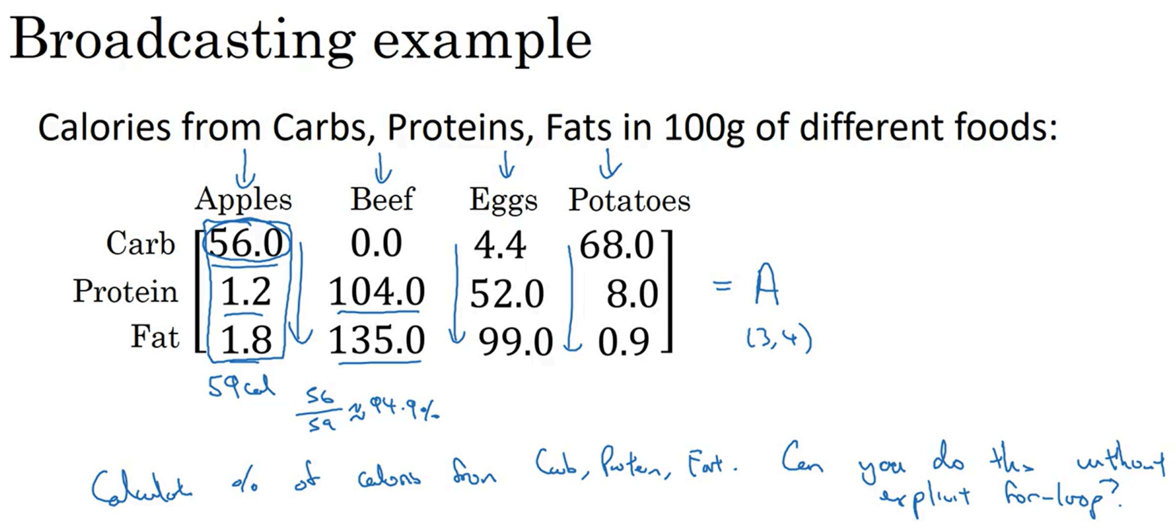
```
for iter in range(1000): ←
    $Z = w^T X + b$
      $= np.dot(w.T, X) + b$
    $A = \sigma(Z)$
    $dZ = A - Y$
    $dw = \frac{1}{m} X dZ^T$
    $db = \frac{1}{m}$ np.sum(dZ)

    $w := w - \alpha dw$
    $b := b - \alpha db$
```

Andrew Ng

Broadcasting turns out to be a technique that Python and numpy allows you to use to make certain parts of your code also much more efficient.

Broadcasting in Python

In the previous video, I mentioned that broadcasting is another technique that you can use to make your Python code run faster. In this video, let's delve into how broadcasting in Python actually works.



Broadcasting example

Calories from Carbs, Proteins, Fats in 100g of different foods:

$$
\begin{array}{c|cccc}
 & \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \\
\hline
\text{Carb} & \boxed{56.0} & 0.0 & 4.4 & 68.0 \\
\text{Protein} & 1.2 & \underline{104.0} & 52.0 & 8.0 \\
\text{Fat} & 1.8 & \underline{135.0} & 99.0 & 0.9
\end{array} = A \quad (3,4)
$$

59cal

$\frac{56}{59} \approx 94.9\%$

Calculate % of calories from Carb, Protein, Fat. Can you do this without explicit for-loop?

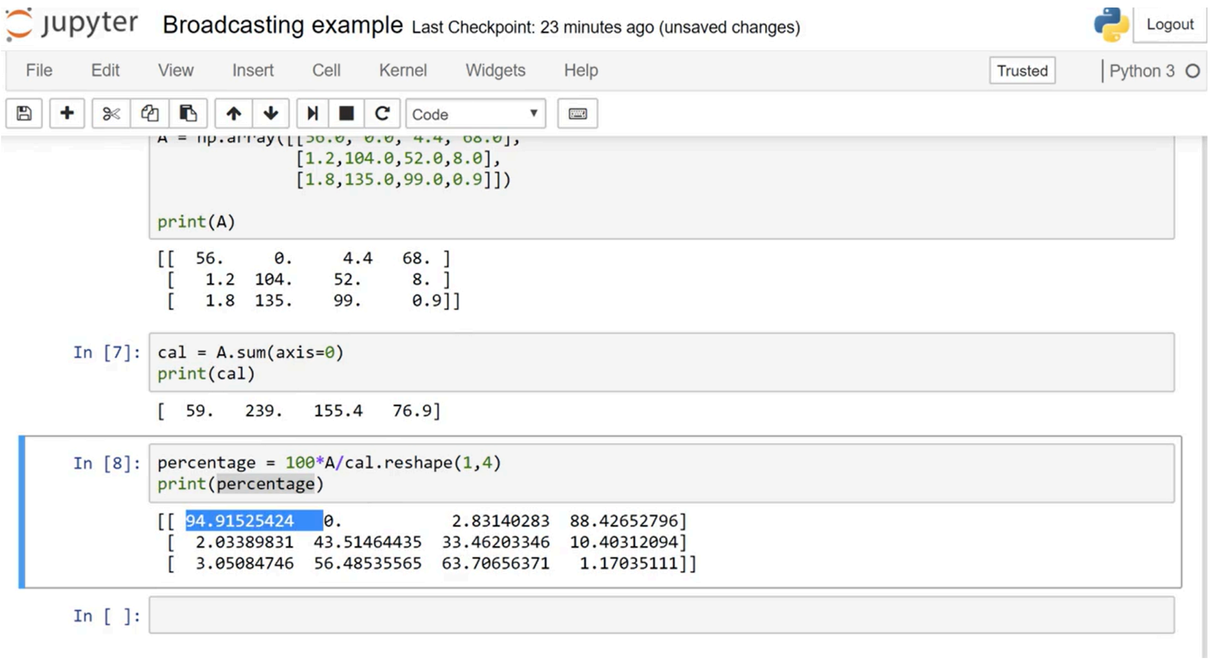
problem:

If you look at this (the first column) column and add up the numbers in that column you get that 100 grams of apple has 56 plus 1.2 plus 1.8 so that's 59 calories.

And so as a percentage the percentage of calories from carbohydrates in an apple would be 56/59, that's about 94.9%. So most of the calories in an apple come from carbs, whereas in contrast, most of the calories of beef come from protein and fat and so on.

So the calculation you want is really to sum up each of the four columns of this matrix to get the total number of calories in 100 grams of apples, beef, eggs, and potatoes. And then to divide throughout the matrix, so as to get the percentage of calories from carbs, proteins and fats for each of the four foods.

So the question is, can you do this without an explicit for-loop? Let's take a look at how you could do that.



jupyter
Broadcasting example Last Checkpoint: 23 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Code

```
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)
```

```
[[  56.     0.     4.4   68. ]
 [   1.2  104.    52.     8. ]
 [   1.8  135.    99.     0.9]]
```

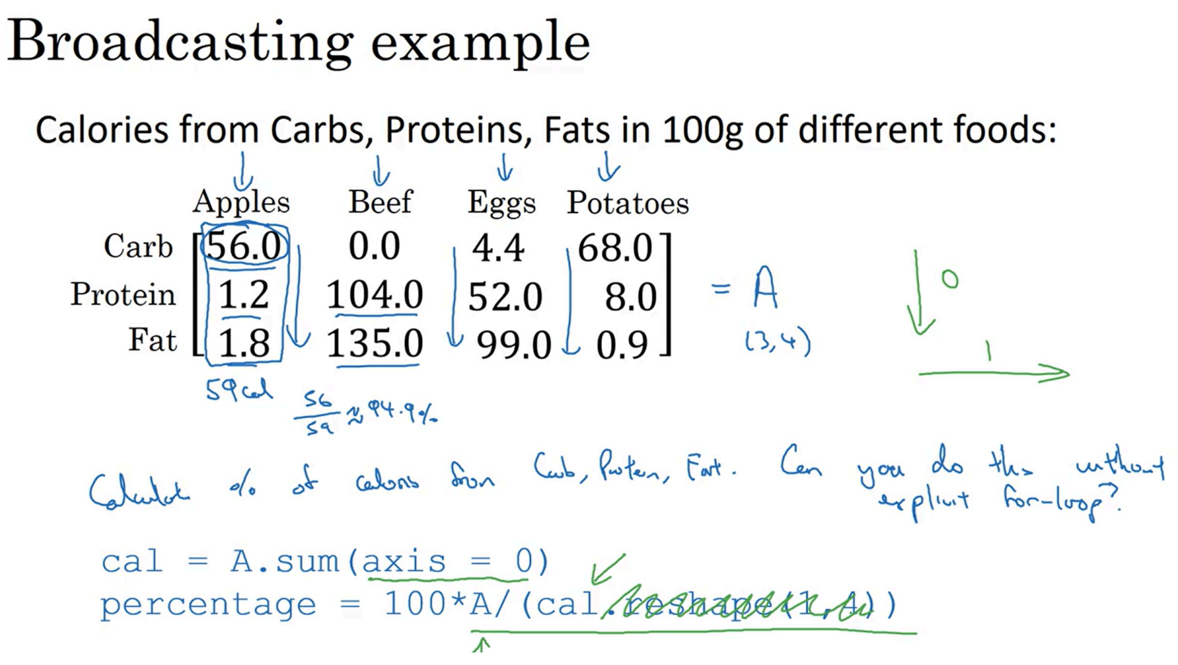
```
In [7]: cal = A.sum(axis=0)
        print(cal)
```

```
[  59.   239.   155.4   76.9]
```

```
In [8]: percentage = 100*A/cal.reshape(1,4)
        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]]
```

```
In [ ]:
```



Broadcasting example

Calories from Carbs, Proteins, Fats in 100g of different foods:

$$\begin{array}{c|cccc} & \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \\ \hline \text{Carb} & \boxed{56.0} & 0.0 & 4.4 & 68.0 \\ \text{Protein} & 1.2 & 104.0 & 52.0 & 8.0 \\ \text{Fat} & 1.8 & 135.0 & 99.0 & 0.9 \end{array} = A \quad (3,4)$$

59cal

$\frac{56}{59} \approx 94.9\%$

Calculate % of calories from Carb, Protein, Fat. Can you do this without explicit for-loop?

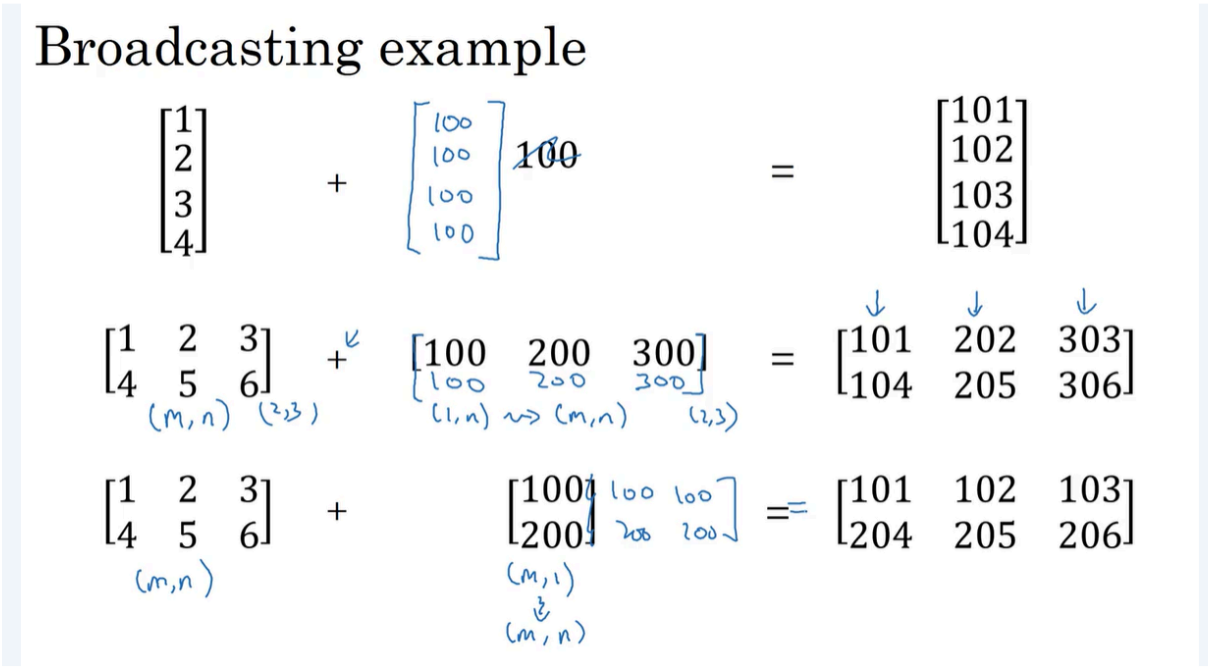
```
cal = A.sum(axis = 0)
percentage = 100*A/(cal~~.reshape(1,4)~~)
```

axis = 0 → sum up vertically

axis = 1 → sum up horizontally

because the shape of ‘cal’ variable is already (1,4) so that we can remove the ‘reshape’ term.

other broadcasting example:

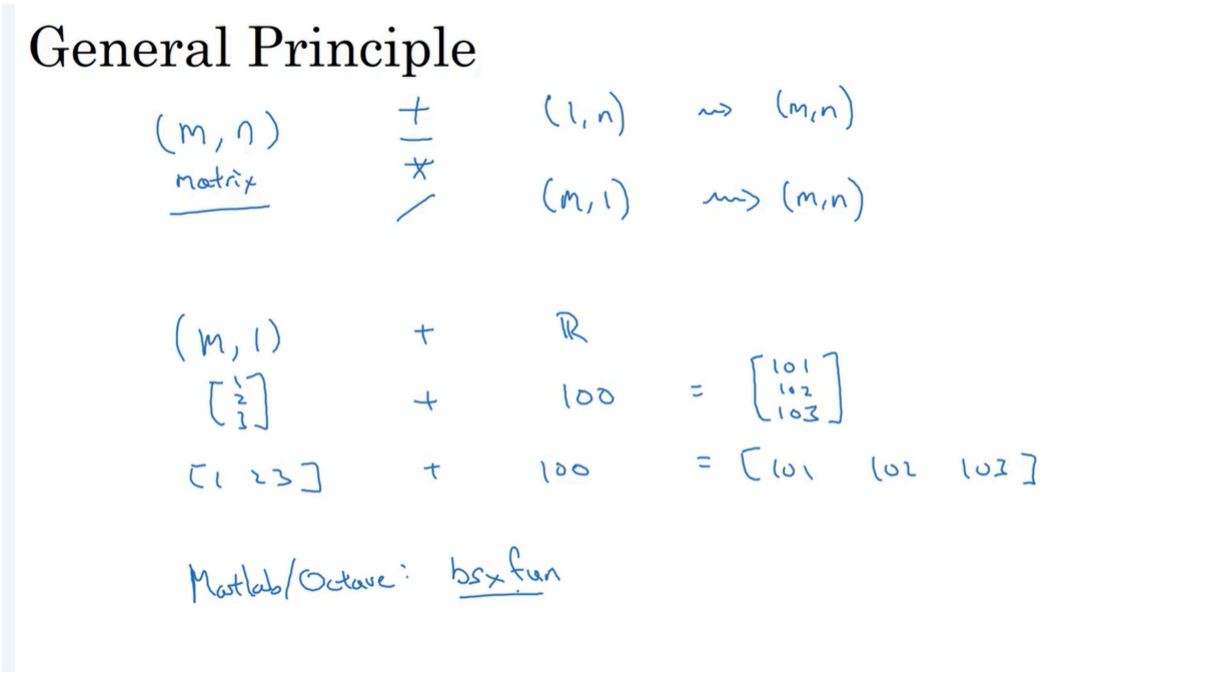


Broadcasting example

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + \begin{bmatrix}100\\100\\100\\100\end{bmatrix} \cancel{100} = \begin{bmatrix}101\\102\\103\\104\end{bmatrix}$$

$$\underset{(m,n)\ (2,3)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \underset{(1,n) \rightsquigarrow (m,n)\quad (2,3)}{\begin{bmatrix}100 & 200 & 300\\100 & 200 & 300\end{bmatrix}} = \begin{bmatrix}101 & 202 & 303\\104 & 205 & 306\end{bmatrix}$$

$$\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \underset{(m,1) \rightsquigarrow (m,n)}{\begin{bmatrix}100 & 100 & 100\\200 & 200 & 200\end{bmatrix}} = \begin{bmatrix}101 & 102 & 103\\204 & 205 & 206\end{bmatrix}$$



General Principle

$$\underset{\text{matrix}}{(m,n)} \quad \begin{matrix}+\\-\\*\\/\end{matrix} \quad \begin{matrix}(1,n) \rightsquigarrow (m,n)\\(m,1) \rightsquigarrow (m,n)\end{matrix}$$

$$(m,1) + \mathbb{R}$$

$$\begin{bmatrix}1\\2\\3\end{bmatrix} + 100 = \begin{bmatrix}101\\102\\103\end{bmatrix}$$

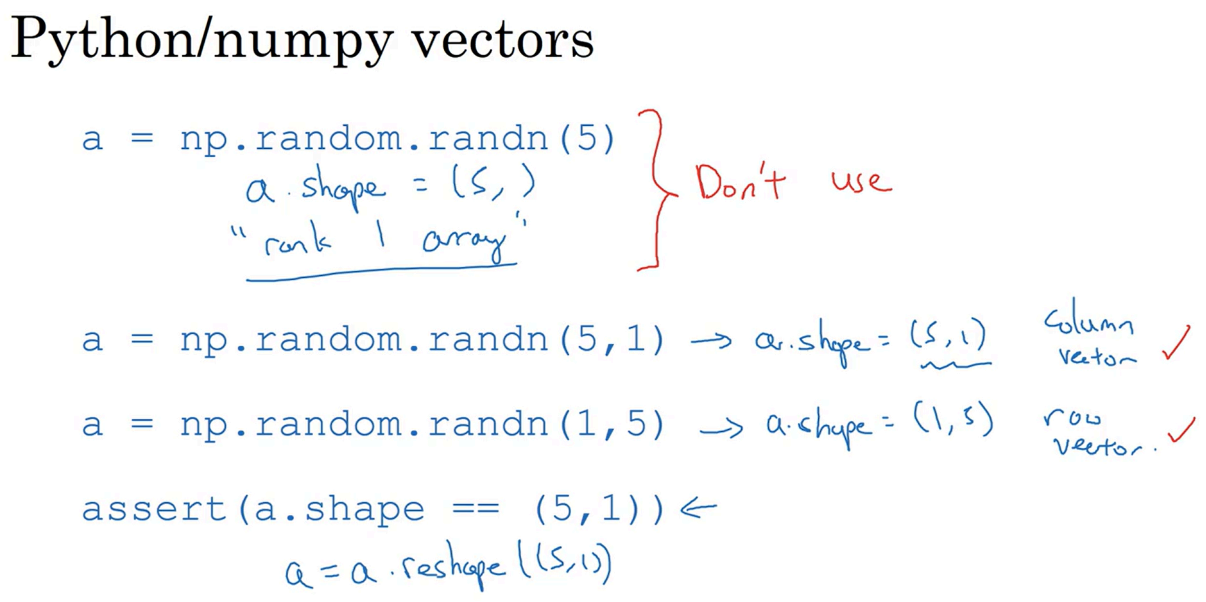
$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} + 100 = \begin{bmatrix}101 & 102 & 103\end{bmatrix}$$

Matlab/Octave: bsxfun

Before we wrap up, just one last comment, which is for those of you that are used to programming in either MATLAB or Octave, if you've ever used the MATLAB or Octave function bsxfun in neural network programming bsxfun does something similar, not quite the same. But it is often used for similar purpose as what we use broadcasting in Python for. But this is really only for very advanced MATLAB and Octave users, if you've not heard of this, don't worry about it. You don't need to know it when you're coding up neural networks in Python. So, that was broadcasting in Python.

I hope that when you do the programming homework that broadcasting will allow you to not only make a code run faster, but also help you get what you want done with fewer lines of code.

A Note on Python/Numpy Vectors



Python/numpy vectors

```
a = np.random.randn(5)
      a.shape = (5, )
      "rank 1 array"
```

Don't use

```
a = np.random.randn(5,1) → a.shape = (5,1)
a = np.random.randn(1,5) → a.shape = (1,5)
assert(a.shape == (5,1)) ←
      a = a.reshape((5,1))
```

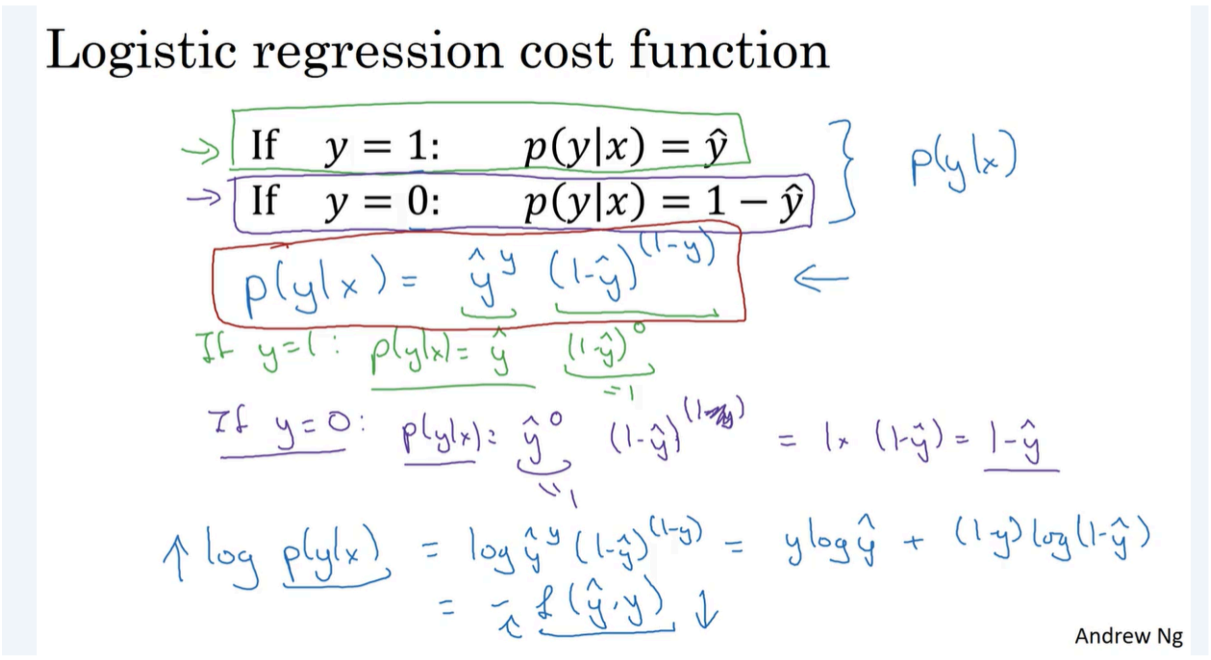
column vector ✓

row vector. ✓

One more thing that I do a lot in my code is if I'm not entirely sure what's the dimension of one of my vectors, I'll often throw in an assertion statement like this, to make sure, in this case, that this is a (5,1) vector. So this is a column vector.

These assertions are really inexpensive to execute, and they also help to serve as documentation for your code. So don't hesitate to throw in assertion statements like this whenever you feel like. And then finally, if for some reason you do end up with a rank 1 array, You can reshape this, a equals a.reshape into say a (5,1) array or a (1,5) array so that it behaves more consistently as either column vector or row vector. So I've sometimes seen students end up with very hard to track because those are the nonintuitive effects of rank 1 arrays. By eliminating rank 1 arrays in my old code, I think my code became simpler. And I did not actually find it restrictive in terms of things I could express in code. I just never used a rank 1 array.

Explanation of Logistic Regression Cost Function



Logistic regression cost function

$$\left.\begin{aligned} &\rightarrow \boxed{\text{If } \; y = 1: \quad p(y|x) = \hat{y}} \\ &\rightarrow \boxed{\text{If } \; y = 0: \quad p(y|x) = 1 - \hat{y}} \end{aligned}\right\} \quad p(y|x)$$

$$\boxed{p(y|x) = \underbrace{\hat{y}^{y}}_{} \; \underbrace{(1-\hat{y})^{(1-y)}}_{}} \quad \leftarrow$$

$$\text{If } y=1: \quad p(y|x) = \hat{y} \quad \underbrace{(1-\hat{y})^{0}}_{=1}$$

$$\text{If } y=0: \quad p(y|x) = \underbrace{\hat{y}^{0}}_{=1} \quad (1-\hat{y})^{(1-0)} \quad = 1 \times (1-\hat{y}) = \underline{1-\hat{y}}$$

$$\uparrow \log \underline{p(y|x)} = \log \hat{y}^{y} (1-\hat{y})^{(1-y)} = y \log \hat{y} + (1-y) \log (1-\hat{y})$$

$$= -\underline{\mathcal{L}(\hat{y}, y)} \downarrow$$

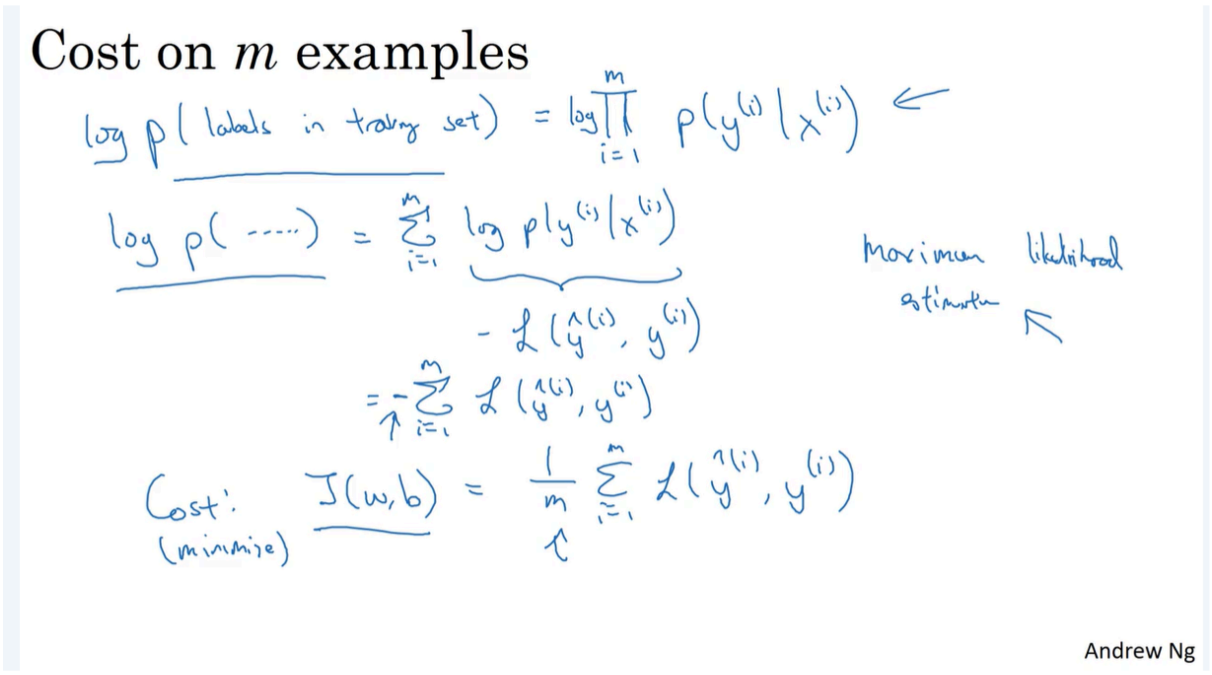
Andrew Ng

So what we've just shown is that this equation is a correct definition for p(ylx).

Now, finally, because the log function is a strictly monotonically increasing function, your maximizing log p(y|x) should give you a similar result as optimizing p(y|x).

And if you compute log of p(y|x), that’s equal to log of y hat to the power of y, 1 - y hat to the power of 1 - y. And so that simplifies to y log y hat + 1- y times log 1- y hat, right?

And so this is actually negative of the loss function that we had to find previously. And there's a negative sign there because usually if you're training a learning algorithm, you want to make probabilities large whereas in logistic regression we're expressing this. We want to minimize the loss function.



Cost on m examples

$$\underline{\log p(\text{labels in training set})} = \log \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}) \leftarrow$$

$$\underline{\log p(\,\cdots\,)} = \sum_{i=1}^{m} \underbrace{\log p(y^{(i)} \mid x^{(i)})}_{-\mathcal{L}(\hat{y}^{(i)},\, y^{(i)})}$$

$$= -\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

maximum likelihood estimation ↖

Cost: (minimize) $\underline{J(w,b)} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

Andrew Ng