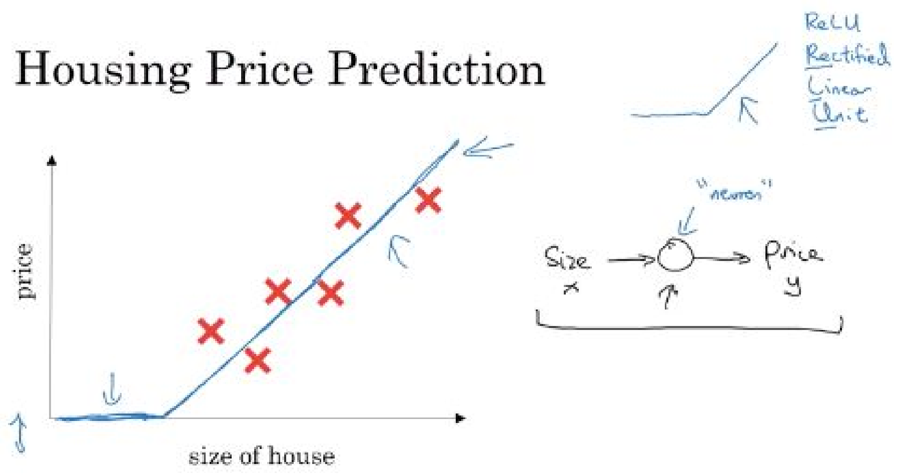
Andrew Ng’s Deep Learning Class

Week 1: Introduction

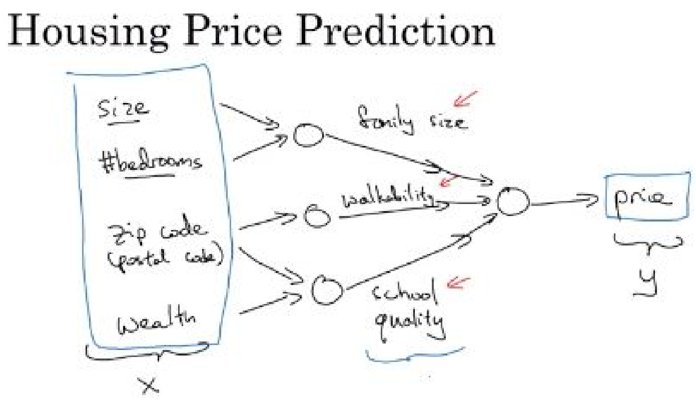
What is a Neural Network?

- Deep Learning refers to training Neural Networks. We are going to talk about Neural Networks in this session. Consider the housing prediction example (given the size of a house we need to predict the price) where we might have a training set with say, 6 houses and you fit a linear regression model to the data. Looks pretty good but we know house prices are not negative so bend the curve at 0 and end up with a function as below. This function is a simple neural network where the entry is the size x, it goes into a neuron and the ‘exit’ is the price y.

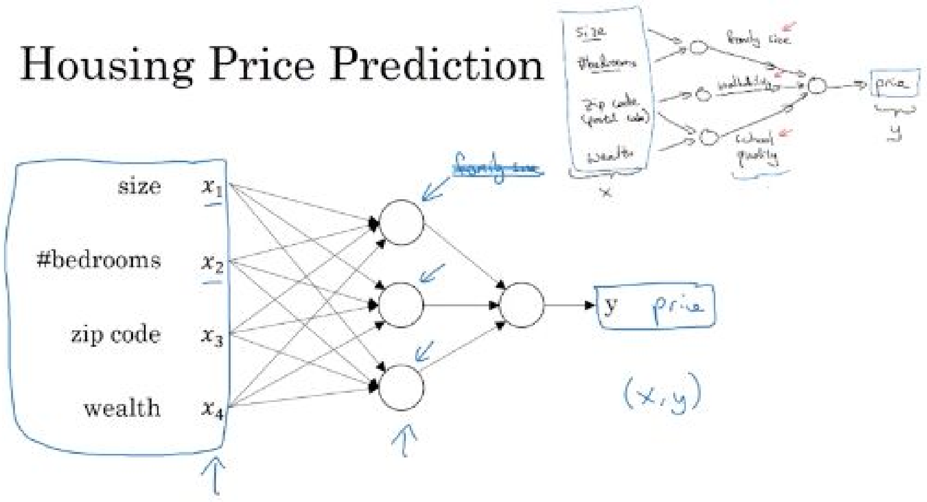


The function we created is seen very frequently in neural networks and is called Rectified Linear Unit (RELU). We can think of the one neuron neural network as a lego brick and we can stack up multiple legos together to create more complex neural networks.

- Let’s look at such an example: Assume we have the price of the house but not only that, we also have # bedrooms and those 2 factors are inputs into a neuron that figures our family size. We also have zip code which as an input to another neuron figures out walkability. Finally, zip code and wealth are inputs into yet another neuron and that neuron’s output is school quality. The 3 outputs (family size, walkability and school quality) then predict the price of the house. In these example, the 4 inputs are x and the output, the price of the house, is y.



- What we end up implementing is then shown below. Notice a couple of things
    - All inputs connect to every single neuron. Instead of us limiting the inputs to a neuron we let the neural network figure it out itself. We just give you all 4 inputs and you, neural network, compute what you want.
    - The middle layer, called the hidden layer, is densely connected to the input layers because every input is connected to every neuron in the middle.

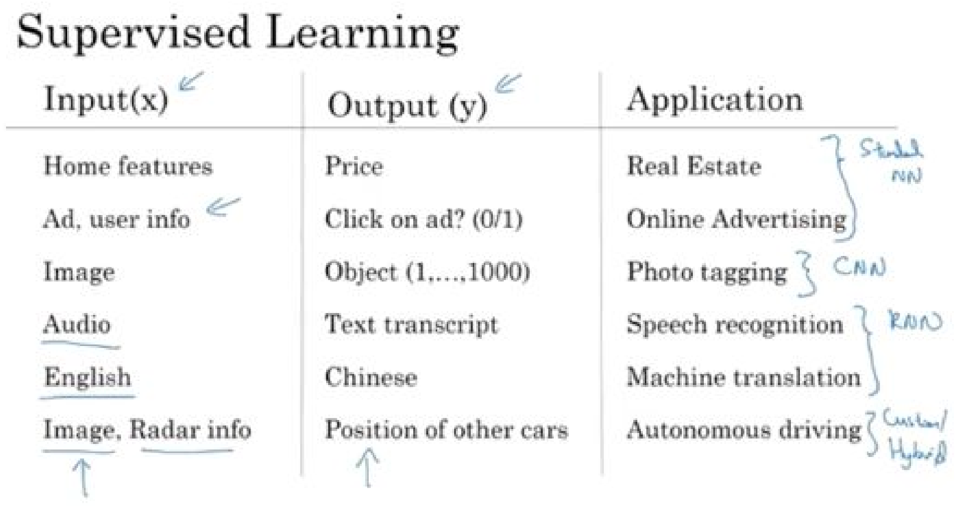


Neural Networks, given enough data, are remarkably good at figuring out functions that map from x to y. These networks are most powerful in in supervised learning settings.

Supervised Learning with Neural Networks

- Most of the economic value created by Machine Learning has been in Supervised Learning. Let's see what type of Supervised Learning applications are using Neural Networks. Some examples are:
    - Real Estate: predicting the price of the house: Standard Neural Network
    - Online Advertising: Standard Neural Network
    - Photo Tagging: Convolutional Neural Network (CNN)

- Speech Recognition: Recurrent Neural Network (RNN)
- Machine Translation: Recurrent Neural Network (RNN)
- Autonomous Driving: Custom/Hybrid Neural Network

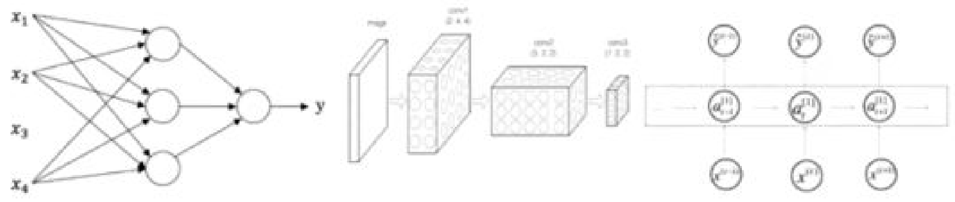


Supervised Learning

Input(x)	Output (y)	Application	
Home features	Price	Real Estate	Stand NN
Ad, user info	Click on ad? (0/1)	Online Advertising	
Image	Object (1,…,1000)	Photo tagging	CNN
Audio	Text transcript	Speech recognition	RNN
English	Chinese	Machine translation	
Image, Radar info	Position of other cars	Autonomous driving	Custom/Hybrid

- How do the different types of neural networks look? We can see below. Standard NN is familiar since it is the one type we studied in the first ML class

Neural Network examples



- We might have heard about machine learning applications for structured data & unstructured data.

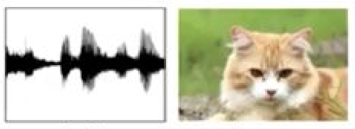
Supervised Learning

Structured Data

Size	#bedrooms	…	Price (1000$s)
2104	3		400
1600	3		330
2400	3		369
⋮	⋮		⋮
3000	4		540

User Age	Ad Id	…	Click
41	93242		1
80	93287		0
18	87312		1
⋮	⋮		⋮
27	71244		1

Unstructured Data



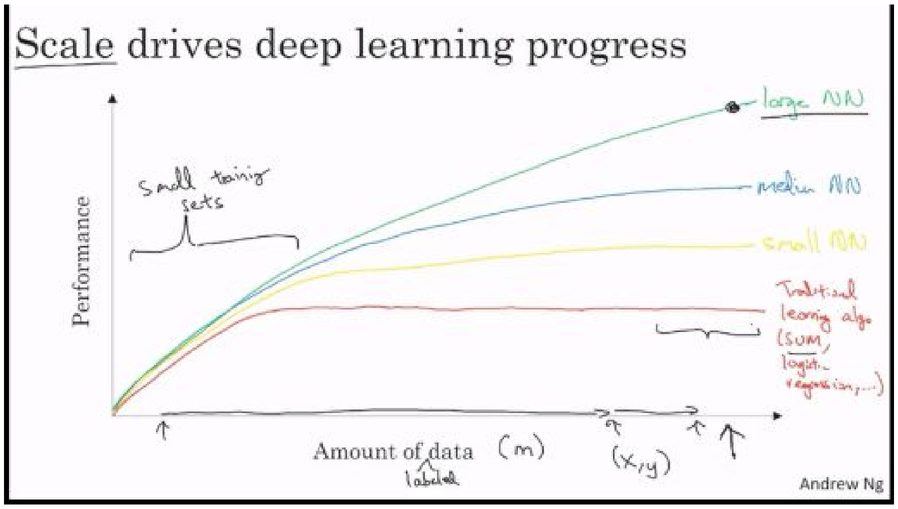
Four scores and seven years ago…

Text

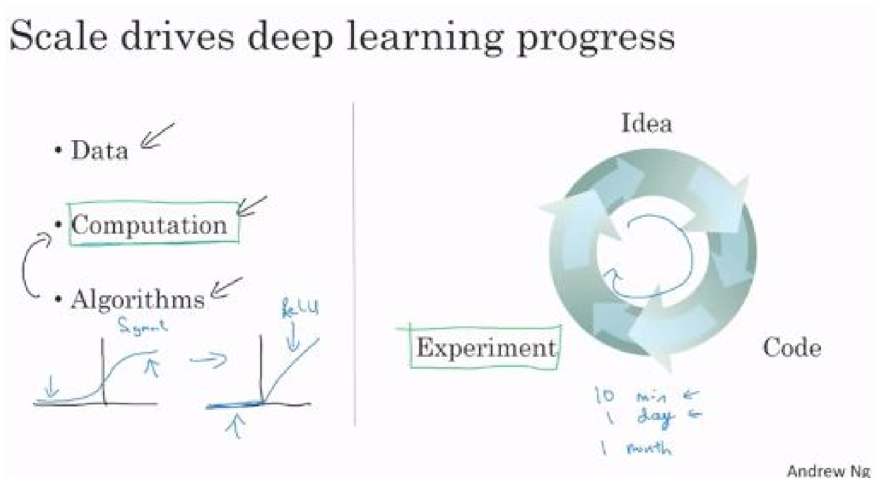
Structured data is basically databases (like the real estate example) where each column has a very well defined meaning. Unstructured data is stuff like images, audio and text where the features might be pixels in an image or words in a piece of text. Computers are much better at interpreting unstructured data thanks to neural networks. However, most of the short term value created by neural networks has been in supervised learning.

Why is Deep Learning taking off right now?

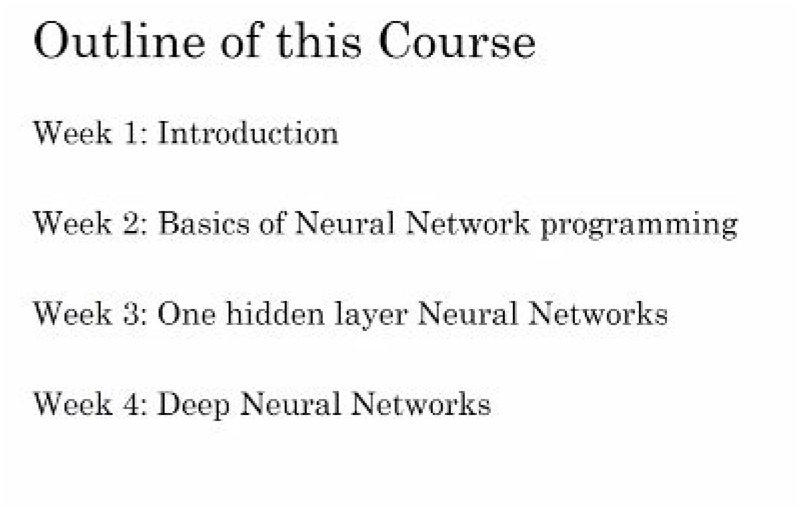
- The basic ideas of neural networks have been around for decades, why is it taking off right now? Well, because we are living on an era where digitization of our activities has led to an explosion of data, so much data that a traditional learning algorithm (SVM, logistic regression) didn't know what to do with it. It turns out a neural network keeps getting better and better as you throw more data at it. We then say that scale is driving neural network progress.



  Today one of the most reliable ways of getting better neural network performance is either train a bigger network or throw more data at it (up to a point). A point we have to make is the amount of data is amount of labeled data (x,y pairs). In the small training set side of the chart above it is not clear the order of the algorithms but as we go to bigger datasets neural networks are the clear winners.
- While data and computation has helped the advancement of neural networks, we have had various innovations in algorithms; for example, moving from a sigmoid function to a RELU has helped gradient descent to work much faster. Additionally, computational advancements has helped because training a neural network is an iterative cycle and it goes faster if we are able to train our network in say 10 minutes or a day instead of a month. Faster computation then has helped us to experiment and iterate much faster than before.



- Finally, our course outline is shown below



Outline of this Course

Week 1: Introduction

Week 2: Basics of Neural Network programming

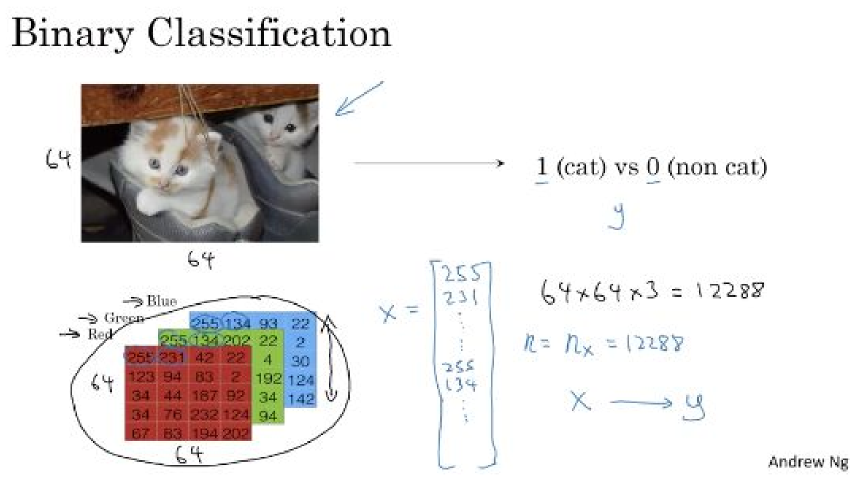
Week 3: One hidden layer Neural Networks

Week 4: Deep Neural Networks

Week 2: Basics of Neural Network Programming

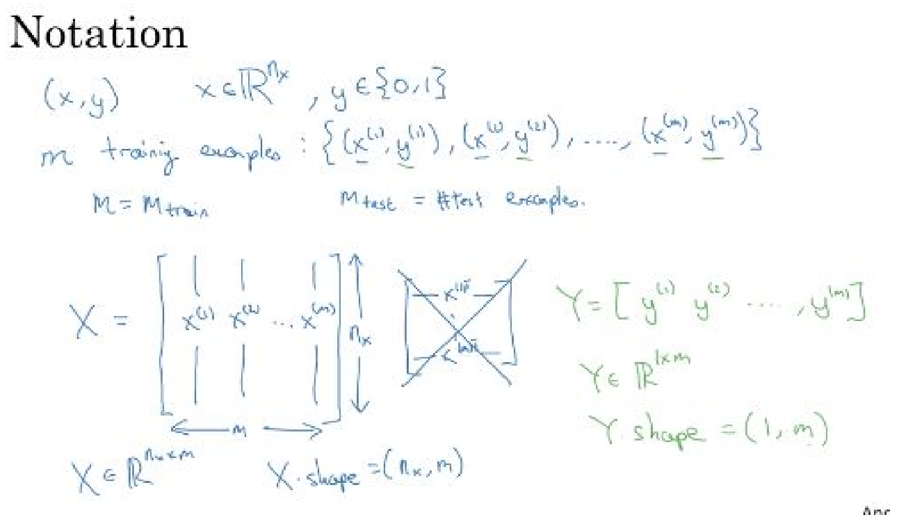
Logistic Regression as a Neural Network: Binary Classification

- We are going to start with Logistic Regression since that will make it easier to understand the main ideas. Logistic regression is an algorithm for binary classification. For example, it would output 1 for the image below if that image corresponds to a cat and it would output 0 otherwise. The input to the algorithm is the 'unrolled' version of the red, green and blue matrices that constitute the image. In the example below we have 3 matrices of 64 x 64 so our 'unrolled' input vector x has a length $n_x = 12288$.



Our goal in binary classification is then to learn a classifier that takes as an input an image represented as an x vector and outputs a prediction of whether or not (1 or 0) the image correspond to a cat (for this example).

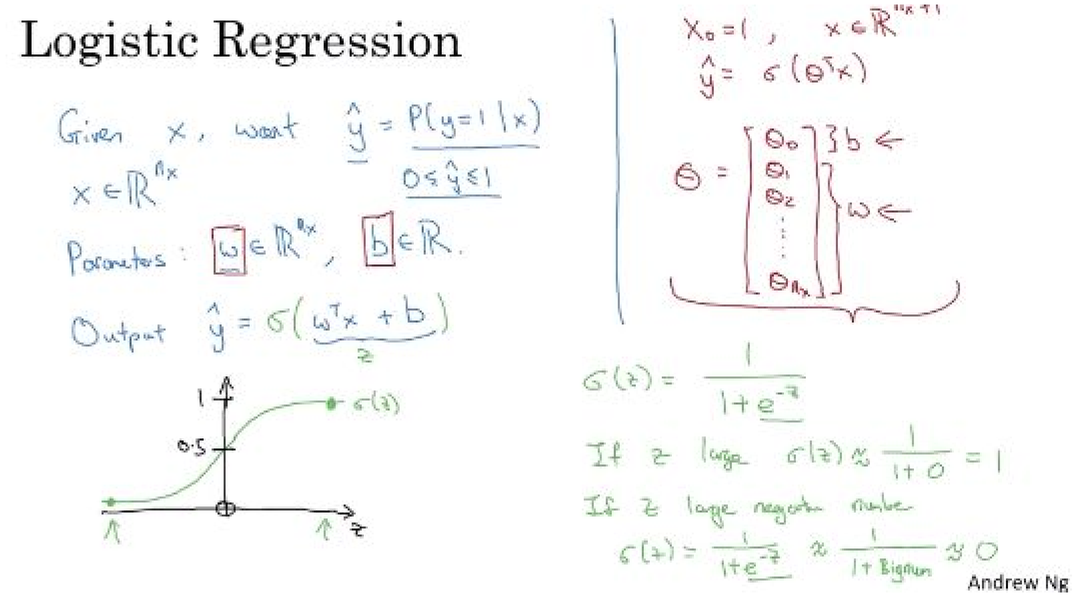
- The notation we are going to use is shown below. We have
  - (x, y) will denote a training example, x has a length n_x and y values are $\{0, 1\}$
  - $m\ = number\ of\ training\ examples$, $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}),\ ...\ (x^{(m)}, y^{(m)})\}$
  - We represent the number of examples in the training set as m_{train} and the number of examples in the test set as m_{test}
  - Finally, we represent all the X examples as a matrix of dimensions $n_x * m$ and Y examples as vector of dimensions $1 * m$. In python, the command would be $x.shape\ = (n_x, m)$ and for you it would be $y.shape = (1, m)$


Notation
(x,y)   x ∈ ℝ^{n_x}, y ∈ {0,1}
m training examples : {(x^(1), y^(1)), (x^(2), y^(2)), ..., (x^(m), y^(m))}
M = M_train   M_test = #test examples.
X =
x^(1) x^(2) ... x^(m)
n_x
m
X ∈ ℝ^{n_x × m}   X.shape = (n_x, m)
Y = [y^(1) y^(2) ... y^(m)]
Y ∈ ℝ^{1×m}
Y.shape = (1, m)

Logistic Regression

- We use logistic regression when the output values y are (0,1). So given X, we want to find out $y^*\ = P(y = 1|\ x)$. The logistic regression parameters are W , which is a vector of

$1\ x\ n_x$ dimensions and b, which is a real number. Our output $y = \sigma(W^T * X + b)$. The sigmoid function is shown on the bottom left corner. So $\sigma(z) = 1/(1 + e^{-z})$. If z is large, $\sigma(z) \approx 1/(1+0) = 1$. If z is a large negative number, $\sigma(z) \approx 1/(1 + very\ large\ number) \approx 0$.


Logistic Regression
Given x, want $\hat{y} = P(y=1 | x)$
$0 \le \hat{y} \le 1$
$x \in \mathbb{R}^{n_x}$
Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$.
Output $\hat{y} = \sigma(\underbrace{w^T x + b}_{z})$
1
0.5
$\sigma(z)$
z
$x_0 = 1$, $x \in \mathbb{R}^{n_x+1}$
$\hat{y} = \sigma(\theta^T x)$
$\theta = [\theta_0, \theta_1, \theta_2, \dots, \theta_{n_x}]$
b
w
$\sigma(z) = \frac{1}{1+e^{-z}}$
If z large $\sigma(z) \approx \frac{1}{1+0} = 1$
If z large negative number
$\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+ \text{Bignum}} \approx 0$
Andrew Ng

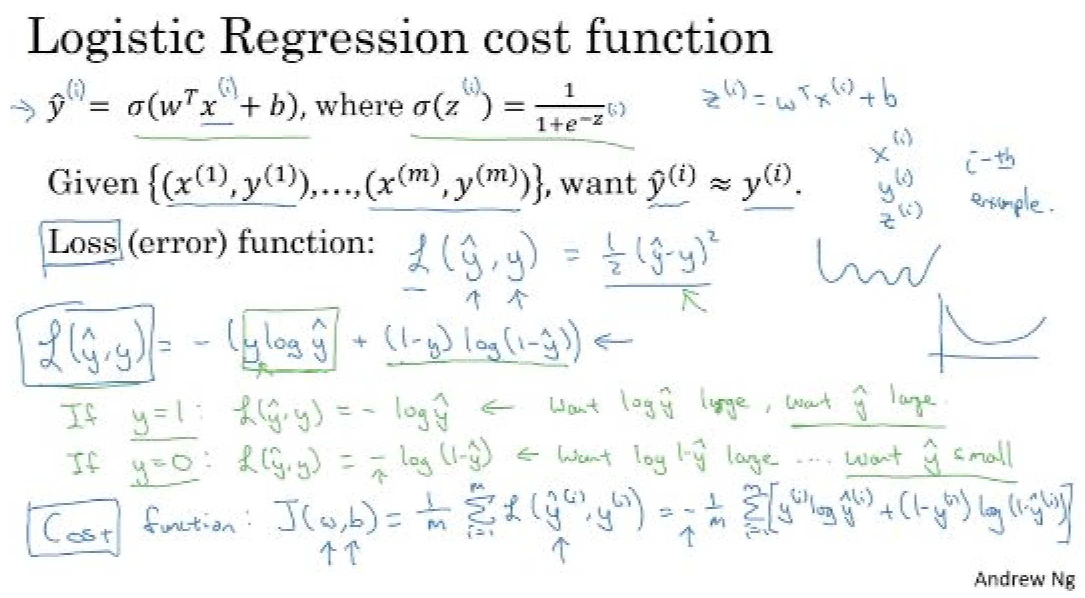
In previous classes we might have used another notation convention (shown in the top right corner) but we are not going to use it here. In that notation we had b and W combined, here we are going to keep them separate.

Logistic Regression Cost Function

- A lot to unpack on this one. We already know the logistic regression equation, now we are going to define the cost function. We want the cost function to be as small as possible because that means that my hypothesis $y*$ is roughly equal to real value y. More formally, we can say that having a training set $\{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$ we want to have $y^{(i)}* \approx y^{(i)}$. We then might be tempted to define the loss (error) function as $L(y^*, y) = (1/2) * (y^* - y)^2$. However, in logistic regression we go with a different loss function: $L(y^*, y) =- (y * log(y^*) + (1-y) * log(1-y^*)$.
  - If y=1 then the loss function is just the first term: $-log(y^*)$ and we want y^* to be very large but since this is the sigmoid function it will never be bigger than 1.
  - If y= then the loss function is just the second term = $log(1-y^*)$ and we want y^* to be very small but it will never be smaller than 0.

  Notice the loss function works on 1 example of the training set. For the whole training set we define the cost function as

$$J(W,\ b) = (1/m) * \sum_{i=1}^{m} L(y^{*(i)},\ y^{(i)}) = \sum_{i=1}^{m} [y^{(i)} * log(y^{*(i)}) + (1-y) * log(1-y^{*(i)})$$



Logistic Regression cost function

$\rightarrow \hat{y}^{(i)} = \sigma(w^T x^{(i)} + b)$, where $\sigma(z^{(i)}) = \frac{1}{1+e^{-z^{(i)}}}$ $\qquad z^{(i)} = w^T x^{(i)} + b$

Given $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$. $\qquad x^{(i)}\ y^{(i)}\ z^{(i)}$ $\quad$ i-th example.

$\boxed{\text{Loss}}$ (error) function: $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

$\boxed{\mathcal{L}(\hat{y}, y)} = -\left(\boxed{y \log \hat{y}} + (1-y)\log(1-\hat{y})\right) \leftarrow$

If $y = 1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y} \leftarrow$ want $\log \hat{y}$ large, want $\hat{y}$ large.

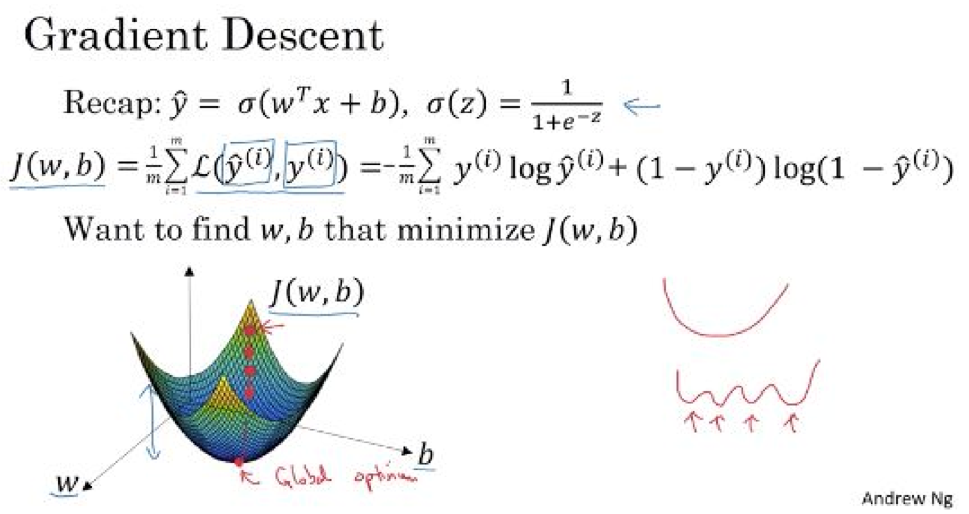
If $y = 0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y}) \leftarrow$ want $\log 1-\hat{y}$ large .... want $\hat{y}$ small

$\boxed{\text{Cost}}$ function: $J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log \hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})\right]$

Andrew Ng

Gradient Descent

- Now we are going to see how we can use gradient descent to learn the parameters W and b. We already defined the cost function $J(W, b)$ and what we want to do is minimize it. This function is convex so it doesn't matter where we start we eventually will converge to the global minimum as shown below.



Gradient Descent

Recap: $\hat{y} = \sigma(w^T x + b)$, $\sigma(z) = \frac{1}{1+e^{-z}} \leftarrow$

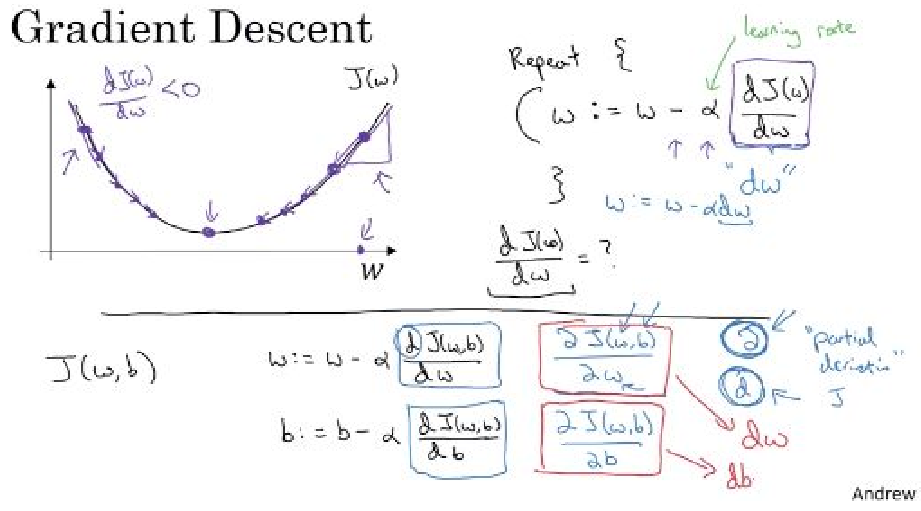
$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\boxed{\hat{y}^{(i)}}, \boxed{y^{(i)}}) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})$

Want to find w, b that minimize $J(w,b)$

$J(w,b)$
b
w
Global optimum

Andrew Ng

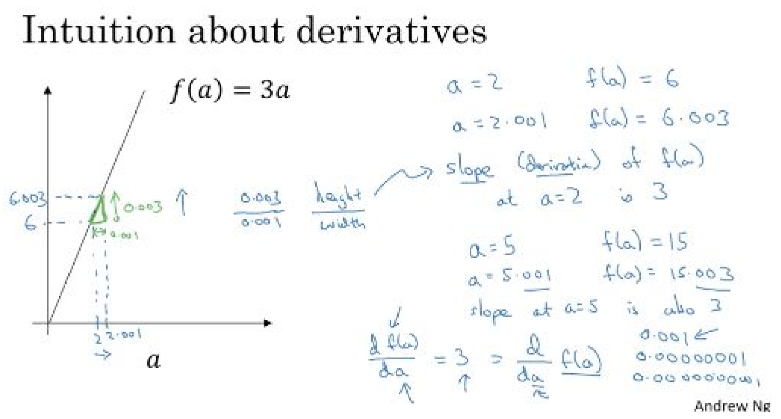
- Let's illustrate this using only W, we have a function $J(W)$ as shown below. Now let's take 2 examples: If we have a large value of $J(W)$ we are at the right of the curve. What we do in every step of gradient descent is to update W as follows: $W := W - \alpha \frac{dJ(w)}{dw}$ where $\frac{dJ(w)}{dw}$ is the derivative which we will represent in our code as variable dw. The definition of a derivative is the slope of the function at that point and the slope is the height/width of the triangle. In the example of the right of the curve that derivative is positive so W gets updated by subtracting the positive derivative so W moves to the left. If we are on the other side of the curve (the left side), the derivative is negative so W gets updated by subtracting a negative derivative which makes it an addition so W moves to the right. This is how gradient descent converges to the global minimum


Gradient Descent
learning rate
Repeat {
$w := w - \alpha \frac{dJ(w)}{dw}$
}
$\frac{dJ(w)}{dw} < 0$
$J(w)$
$w := w - \alpha\, dw$
"dw"
$\frac{dJ(w)}{dw} = ?$
w
$J(w,b)$
$w := w - \alpha \frac{dJ(w,b)}{dw}$
$b := b - \alpha \frac{dJ(w,b)}{db}$
$\frac{\partial J(w,b)}{\partial w}$
$\frac{\partial J(w,b)}{\partial b}$
"partial derivative"
J
dw
db
Andrew Ng

The real cost function is a function of two variables so we have two partial derivatives to worry about: $dw = (d * J(W, b))/dW$ and $db = (d * J(W, b))/db$

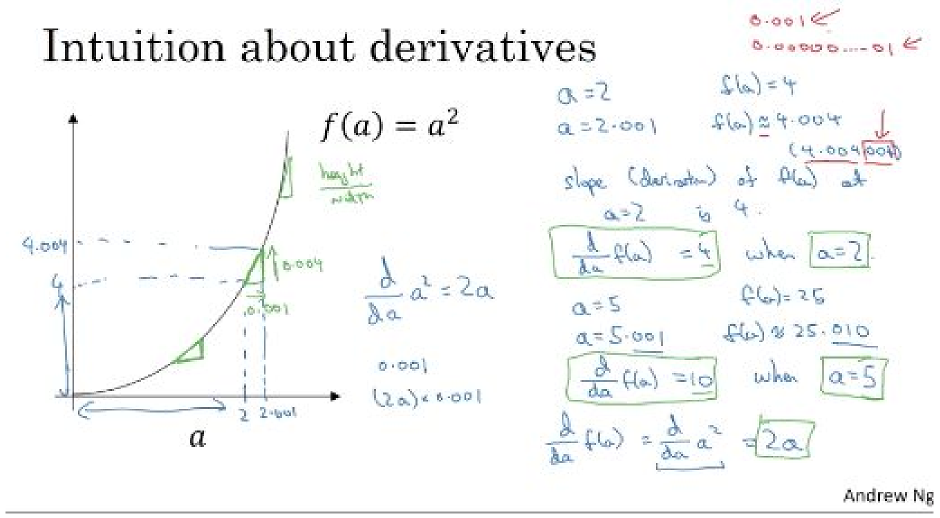
Derivatives

- We are going to explain derivatives as follows: assume we have a function $f(a) = 3a$. Let's say $a = 2$, then $f(a) = 6$ and we plot that value below. Now let's 'nudge' a to the right a tiny bit, 0.001. $f(a)$ *is then* 6.003. What this shows is that we move a to the right (the width) 0.001, f(a) moved up (the height) 0.003. The derivative is then 0.003/0.001=3. If we take another value, say $a = 5$, we end up with the same result, the derivative $df(a)/da = 3$. In a straight like the derivative is constant.

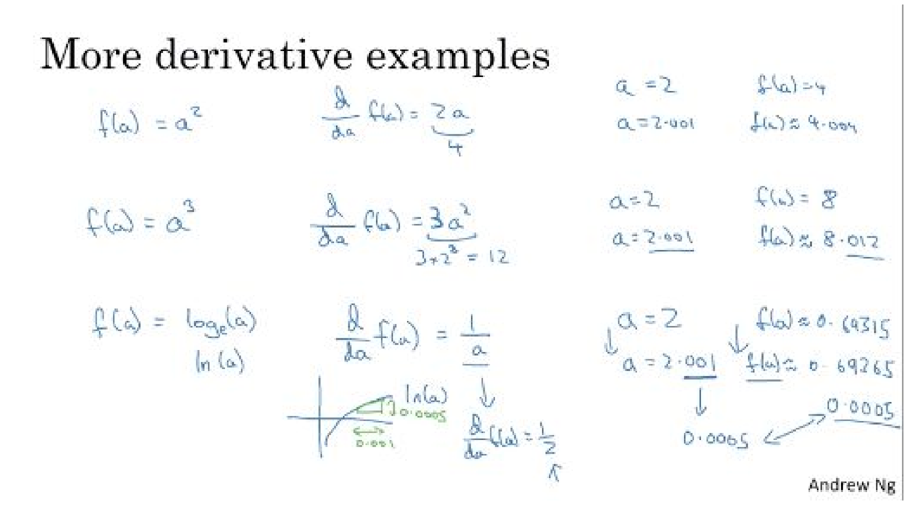

Intuition about derivatives
$f(a) = 3a$
6.003
6
0.003
0.001
$\frac{0.003}{0.001}$
$\frac{\text{height}}{\text{width}}$
a
2
2.001
$a = 2$ $f(a) = 6$
$a = 2.001$ $f(a) = 6.003$
slope (derivative) of f(a) at a=2 is 3
$a = 5$ $f(a) = 15$
$a = 5.001$ $f(a) = 15.003$
slope at a=5 is also 3
$\frac{df(a)}{da} = 3 = \frac{d}{da} f(a)$
0.001
0.00000001
0.000000001
Andrew Ng

More Derivative Examples

- In this video we will see examples where the slope of the function can be different at different points. Let’s take for example $f(a) = a^2$. if we take a=2 then f(a) =4 and nudging a to the right by 0.001 gives us f(a) = 4.004001 which is roughly 0.004 so our derivative is 0.004/001= 4. In mathematical notation we say $df(a)/da = 4 \ when \ a = 2$. If we take another value, a=5 we end up with $df(a)/da = 10 \ when \ a = 5$. A calculus book will tell us that $d\,f(a)/da = da^2/da = 2a$ which we can see is the case.

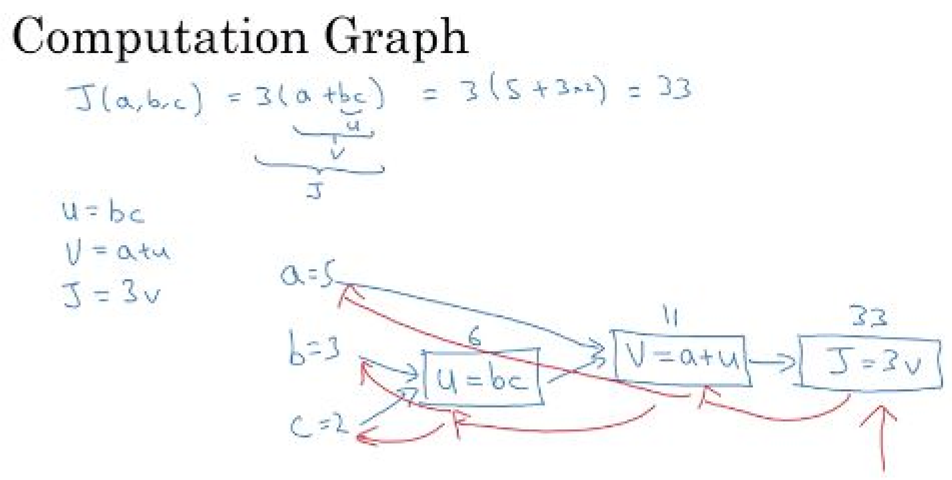


- Now let’s look at a couple of other examples: we already saw $f(a) = a^2$. Now let’s see $f(a) = a^3$ where the derivative $d\,f(a)/da = 3a^2$ and $f(a) = log(a)$ which derivative is $d\,(fa)/da = 1/a$. Our two main takeaways are:
  - The derivative of the function is just the slope of the function
  - If you want to look the derivative of a function you can look it up on a textbook or in wikipedia



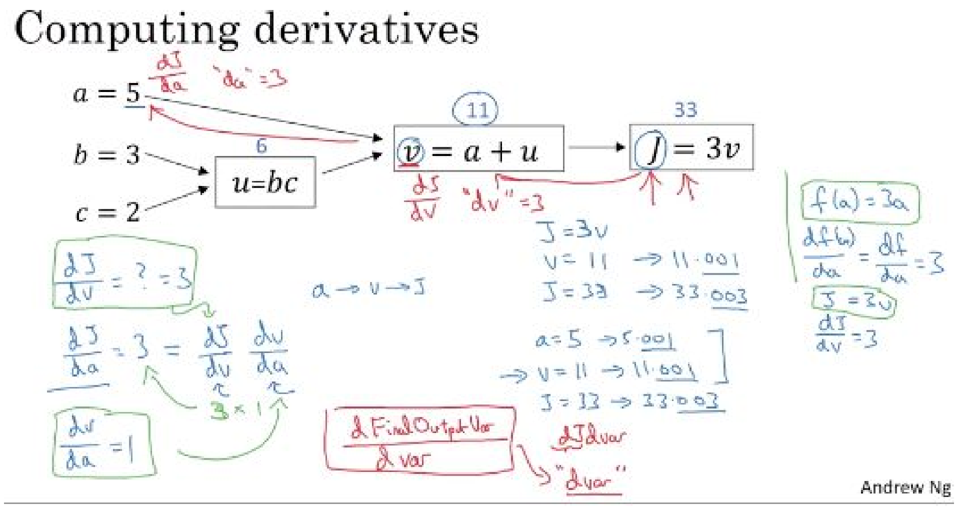
Computation Graph

- An example of a computation graph is shown below. A computation graph is handy when needing to compute a variable such as J which is our cost function. We can see how there are some steps that go from left to right with blue arrows. To compute the derivatives we go from right to left with red arrows. To recap, computation graph organizes the computation from left to right with blue arrows



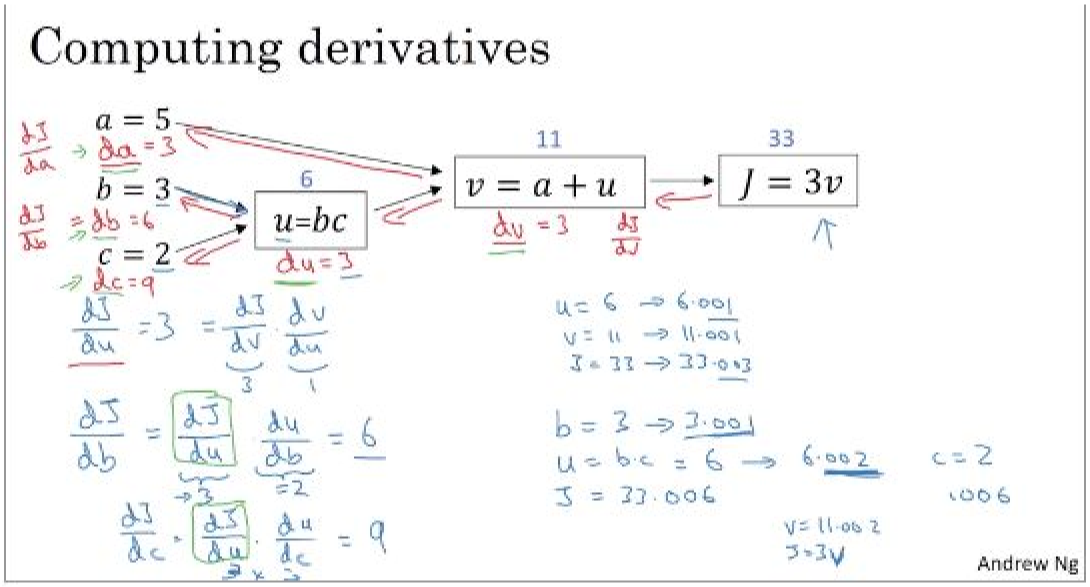
Derivatives with a Computation Graph

- Now let's use a computation graph to see how we can compute derivatives for function J. Below we have the computation graph from the previous class and we are going to compute the derivative for J with respect to v which is represented as dJ/dv. We compute it as follows: if J=3v and v=11 and we nudge v 0.001 we end up with v=11.001 and J=33.003 so the derivative is 0.003/001=3 so $dJ/dv = 3$. We can also compute dJ/da. If a=5 and v=11 and we nudge a 0.001 then a=5.001, v=11.001 and J=33.003. The derivative is then $dJ/da = 3$.



And that's how we use backprop to compute derivatives of J with respect to another var. In our code we are going to denote the derivative of J with respect to a var as $dJ/dvar$ as just 'dvar'.

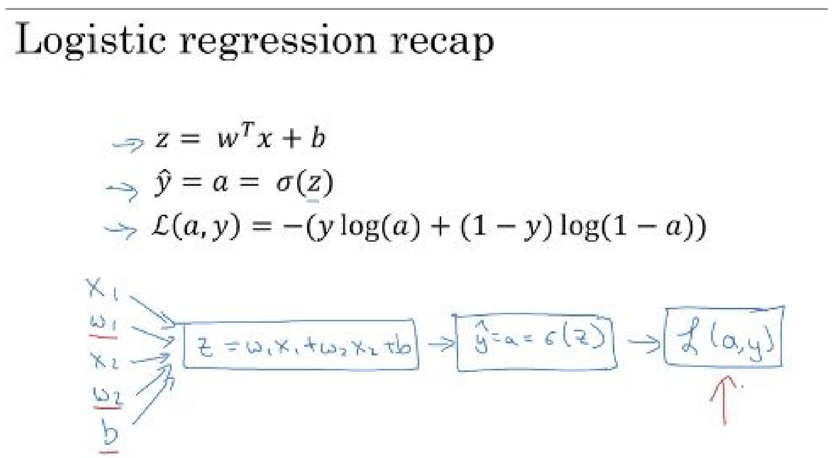
- Let's continue to go backwards. We can compute dJ/db using the calculus chain rule and we have $dJ/db = dJ/du \; * du/db$ .We compute dJ/du as: taking u=6, v=11 and J=33 and nudging u to 6.001 we end up with v=11.001 and with J=33.003 so dJ/du =3. Now we compute and du/db as: if b=3, u = b*c =6 and J =33.006 so $du/db = 2$. So $dJ/db = 3 * 2 = 6$. We are not going to do the whole computation but $dJ/dc = dJ/du \; * \; du/dc = 9$



The key takeaway is that the most efficient way to compute derivatives is to go from right to left following the direction of the red arrows.

Logistic Regression Gradient Descent

- Let's now try to do the derivatives that matter for logistic regression. We know $z = W^T * x + b$ and $y^* = a = \sigma(z)$ and $L(a, y) = -(y * log(a) + (1 - y) * log(1 - a))$. Our computation graph is shown below: all the inputs go to $z = w_1x_1 + w_2x_2 + b$ which then goes into $y^* = a = \sigma(z)$ which then goes into $L(a, y)$



Logistic regression recap

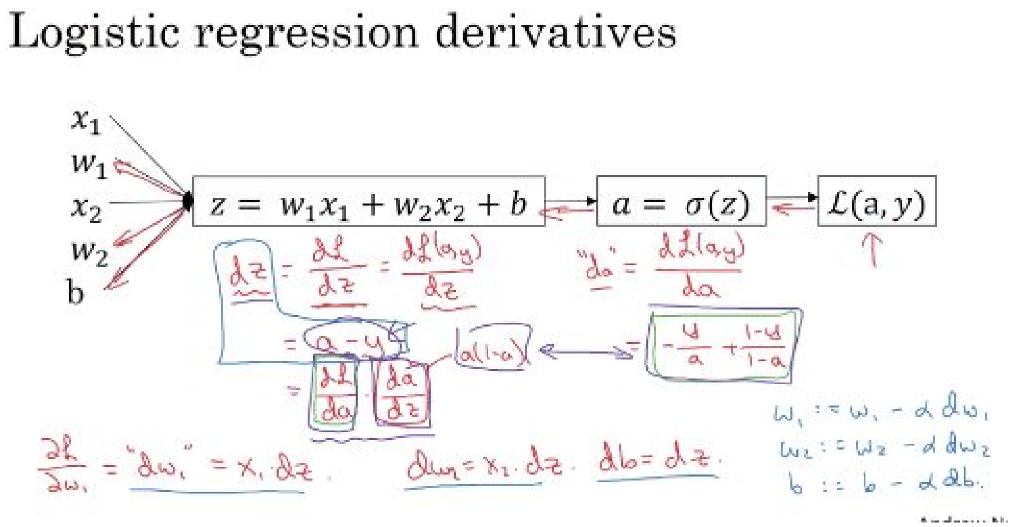
$\rightarrow z = w^T x + b$

$\rightarrow \hat{y} = a = \sigma(z)$

$\rightarrow \mathcal{L}(a, y) = -(y\log(a) + (1-y)\log(1-a))$

- Having our computation graph we can go back and compute our derivatives. They are:
    - $da =-(y/a) + (1-d)/(1-a)$
    - $dz = a - y$
    - $dw_1 = x_1 dz$
    - $dw_2 = x_2 dz$
    - $db = dz$

  And we perform gradient descent as
    - $w_1 = w_1 - \alpha\, dw_1$
    - $w_2 = w_2 - \alpha\, dw_2$
    - $b = b - \alpha\, db$



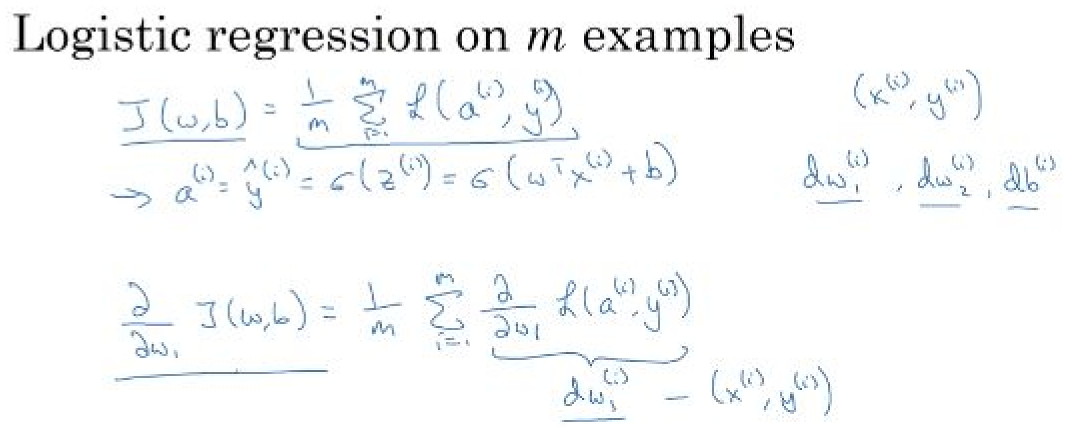
Logistic regression derivatives

x_1
w_1
x_2
w_2
b
$z = w_1x_1 + w_2x_2 + b$
$a = \sigma(z)$
$\mathcal{L}(a, y)$
$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz}$
$= a - y$
$= \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$
$a(1-a)$
"da" $= \frac{d\mathcal{L}(a,y)}{da}$
$= -\frac{y}{a} + \frac{1-y}{1-a}$
$\frac{\partial \mathcal{L}}{\partial w_1} =$ "dw_1" $= x_1 \cdot dz$
$dw_2 = x_2 \cdot dz$
$db = dz$
$w_1 := w_1 - \alpha\, dw_1$
$w_2 := w_2 - \alpha\, dw_2$
$b := b - \alpha\, db$

That was gradient descent on one example. Now let's see how we do it for m examples.

Gradient Descent on m examples

- Let's start by remembering the cost function which is $J(W,b) = (1/m) * \sum_{i=1}^{m} L(a^{(i)}, y^{(i)})$ where $a^{(i)} = y^{*(i)} = \sigma(z^{(i)}) = \sigma(W^T * x^{(i)} + b)$



Logistic regression on m examples

$J(w,b) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})}$

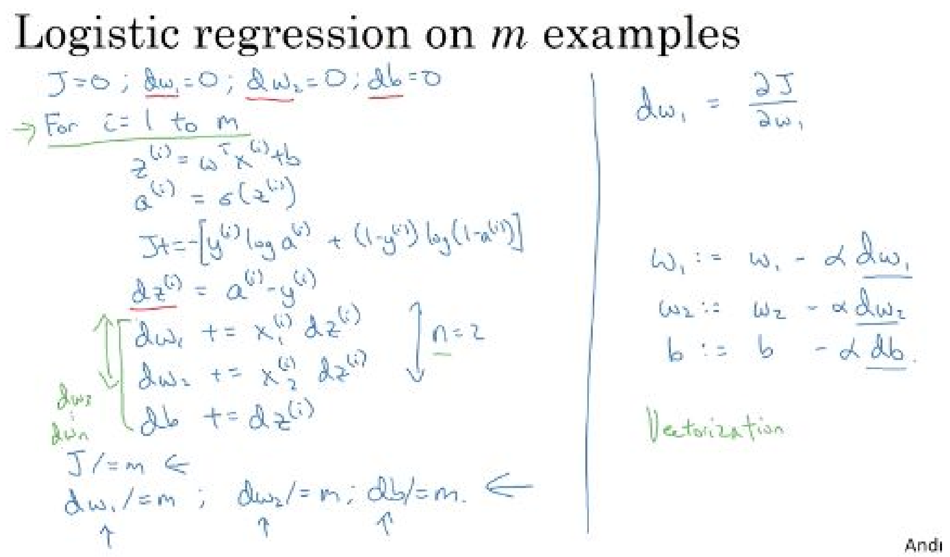
$\to a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$

$(x^{(i)}, y^{(i)})$

$dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

$\frac{\partial}{\partial w_1} J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1}\mathcal{L}(a^{(i)}, y^{(i)})}_{dw_1^{(i)}} - (x^{(i)}, y^{(i)})$

- What is a more concrete way of performing logistic regression on m examples? We start by initializing all our values to zero. We then do a for loop on the m examples where we do the computations shown below and then finally we do the updates to w_1, w_2, b. However, this is only for 1 step of gradient descent. In the era of deep learning it is imperative that we use vectorization to avoid the for loops. It was a nice to have before, now it's mandatory. We will look at vectorization in the next sessions.



Logistic regression on m examples

```
J=0; dw1=0; dw2=0; db=0
For i=1 to m
    $z^{(i)} = w^T x^{(i)} + b$
    $a^{(i)} = \sigma(z^{(i)})$
    $J += -[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})]$
    $dz^{(i)} = a^{(i)} - y^{(i)}$
    $dw_1 += x_1^{(i)} dz^{(i)}$
    $dw_2 += x_2^{(i)} dz^{(i)}$      n=2
    $db += dz^{(i)}$
J /= m
dw1 /= m; dw2 /= m; db /= m
```

$dw_1 = \frac{\partial J}{\partial w_1}$

$w_1 := w_1 - \alpha\, dw_1$

$w_2 := w_2 - \alpha\, dw_2$

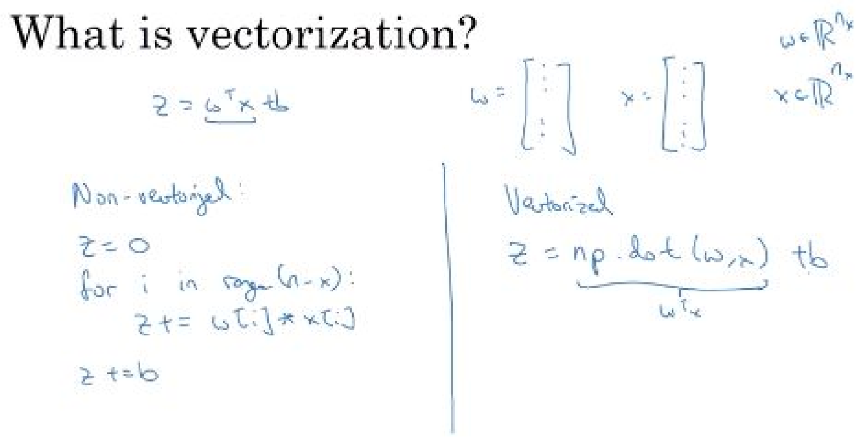
$b := b - \alpha\, db$

Vectorization

dw3
dwn
Andı

Python and Vectorization

- Vectorization is the art of getting rid of for-loops so our code can run faster. On the left side we have a non-vectorized implementation where we have a for-loop over the range of examples; this is very slow. On the right hand we have a vectorized implementation where instead of going row by row we do all the computations in one shot. In python the command to compute $W^T * x$ is $np.\,dot(w, x)$ and then you add b to end with $z = np.\,dot(w, x + b$



What is vectorization?

$z = w^T x + b$

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

Non-vectorized:

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

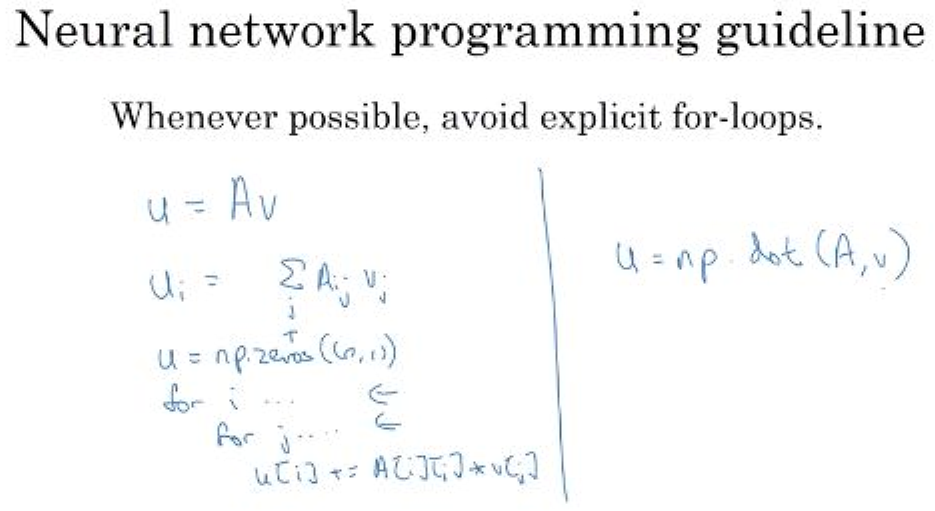
Vectorized

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

- Professor Ng then showed a Jupyter Notebook demo where the vectorized version took about 1.5 ms to run while the non-vectorized version took about 474 ms (around 300 times slower). Whenever possible, we should avoid using for-loops and use a vectorized implementation.

More Vectorization Examples

- Let's take a look at a couple of more vectorization examples. On the left side we have a non-vectorized implementation where we have 2 for-loops. On the right side we have the faster, vectorized implementation that is only 1 line.



Neural network programming guideline

Whenever possible, avoid explicit for-loops.

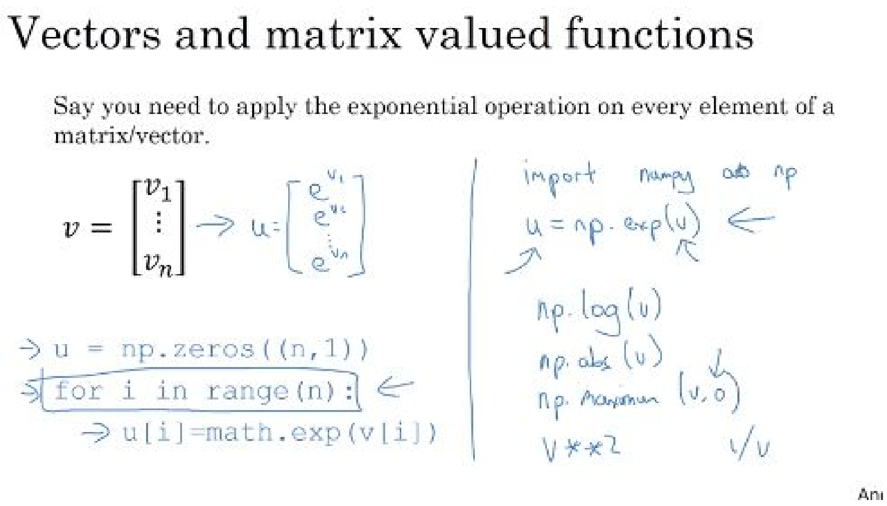
$u = Av$

$u_i = \sum_j A_{ij} v_j$

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

u = np.dot(A,v)

- Below is another example, in this case of applying the exponential function to each element in a vector. Notice how we go from one for-loop and 3 lines of code to 1 line of code. Numpy has many functions that can be applied to vectors, some examples are shown in the right side below.



Vectors and matrix valued functions

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

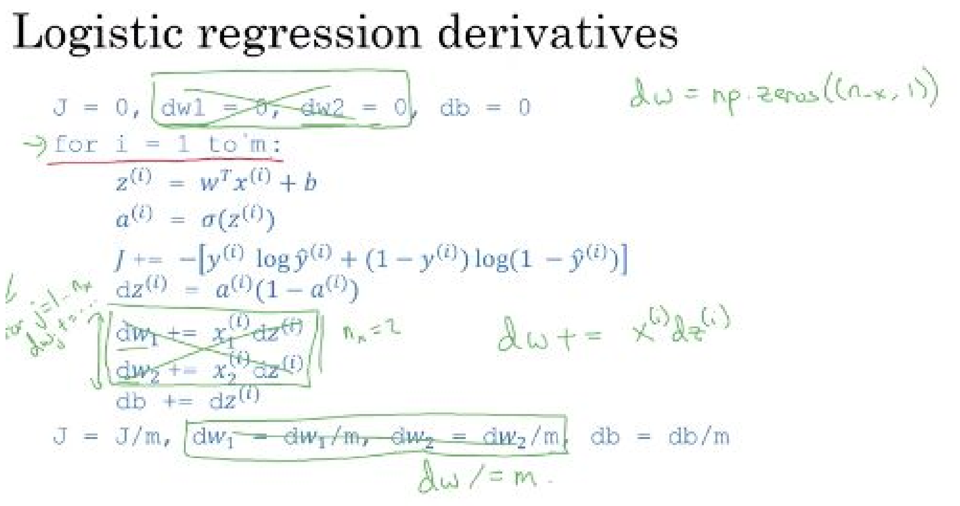
$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \rightarrow u = \begin{bmatrix} e^{v_1} \\ e^{v_2} \\ \vdots \\ e^{v_n} \end{bmatrix}$$

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

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

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

- Now let’s start to simplify our previous explanation of logistic regression. We originally had 2 for-loops and we are going to get rid of the inner loop by substituting with $dw \ += \ x^{(i)} * dz^{(i)}$. In the next session we are going to have a logistic regression implementation with no for-loops.

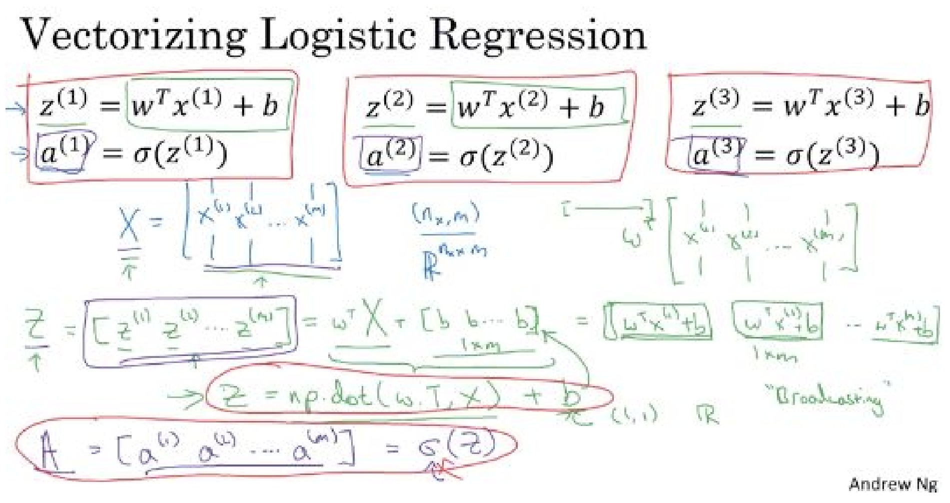


Logistic regression derivatives

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

Vectorizing Logistic Regression

- Below is the vectorized implementation of logistic regression using python commands. Instead of looping through the training set examples one by one what we do is use 2 lines of code to do the whole thing:
  - $Z = np.dot(w.T, X) \ + b$ and
  - $A = \sigma(Z)$ where σ is the sigmoid function call.



Vectorizing Logistic Regression

$z^{(1)} = w^Tx^{(1)} + b$ $\quad$ $z^{(2)} = w^Tx^{(2)} + b$ $\quad$ $z^{(3)} = w^Tx^{(3)} + b$

$a^{(1)} = \sigma(z^{(1)})$ $\quad$ $a^{(2)} = \sigma(z^{(2)})$ $\quad$ $a^{(3)} = \sigma(z^{(3)})$

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

$Z = [z^{(1)}\ z^{(2)} \cdots z^{(m)}] = w^TX + \underbrace{[b\ b \cdots b]}_{1\times m} = \underbrace{[w^Tx^{(1)}+b \quad w^Tx^{(2)}+b \quad \cdots \quad w^Tx^{(m)}+b]}_{1\times m}$

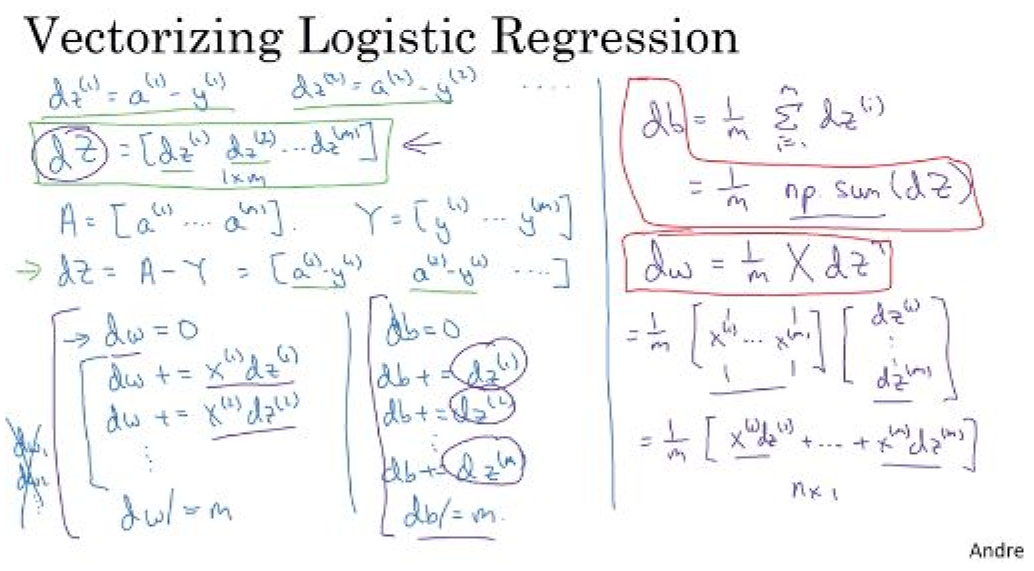
$Z = \text{np.dot}(w.T, X) + b$ $\quad (1,1)$ $\quad \mathbb{R}$ $\quad$ "Broadcasting"

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

Andrew Ng

Vectorizing Logistic Regression's Gradient Computation

- Below are the two lines of code we need to vectorized gradient descent. They are
  - $dz = A = Y$
  - $db = (1/m) * np.sum(dz)$
  - $dw = (1/m) * Xdz^T$ and the resulting vector is nx1 since we are multiplying X (dimensions nxm) and dz (dimensions 1xm) so we need the transpose so the multiplication can happen (nxm * m*1 = nx1)



Vectorizing Logistic Regression

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

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

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

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

$\cancel{dw_1}$ $\cancel{dw_2}$

$dw = 0$

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

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

$\vdots$

$dw /= m$

$db = 0$

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

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

$\vdots$

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

$db /= m$

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

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

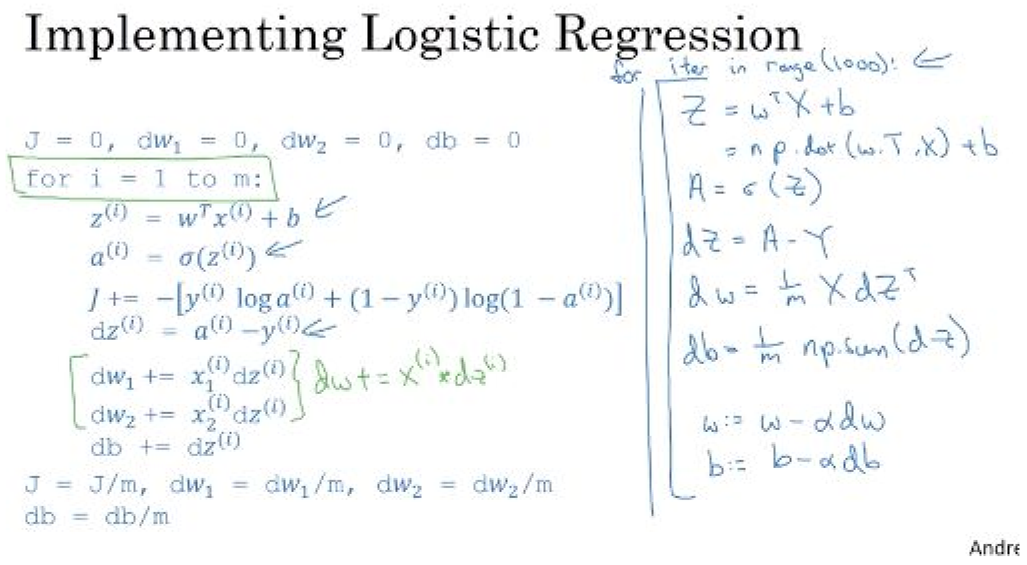
$dw = \frac{1}{m} X dZ^T$

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

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

Andre

- Now we can put everything together to have a fully vectorized version of logistic regression. The version on the right is the vectorized version for 1 iteration of gradient descent. We still need a for-loop to iterate through all the iterations (no way to get rid of that for-loop)



Implementing Logistic Regression

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

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

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

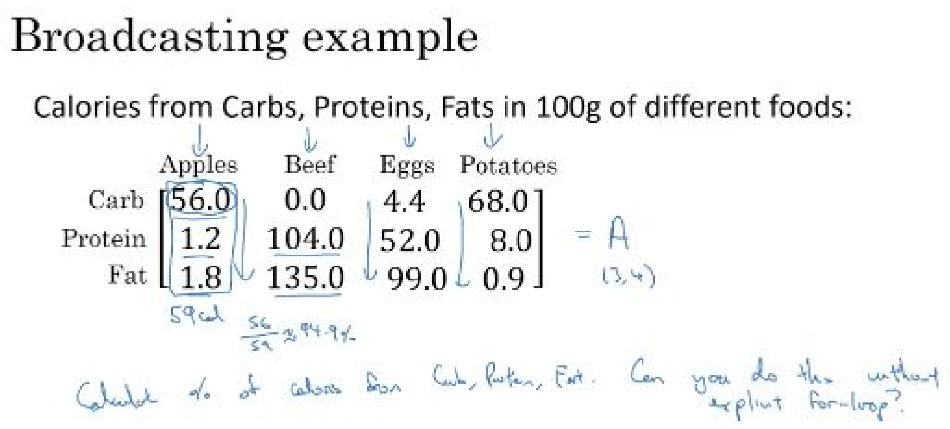
Andre

So the full python version of vectorized logistic regression is:

- $Z = np.dot(w.T, X) + b$
- $A = \sigma(Z)$
- *for number of iterations*
  - $dz = A = Y$
  - $db = (1/m) * np.sum(dz)$
  - $dw = (1/m) * X dz^T$
- $w := w - \alpha * dw$
- $b := b - \alpha * db$

Broadcasting in Python

- We are going to use an example to illustrate how broadcasting works. We have matrix A and we want to know the % of calories for each component. Can we do it without a for?



Broadcasting example

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

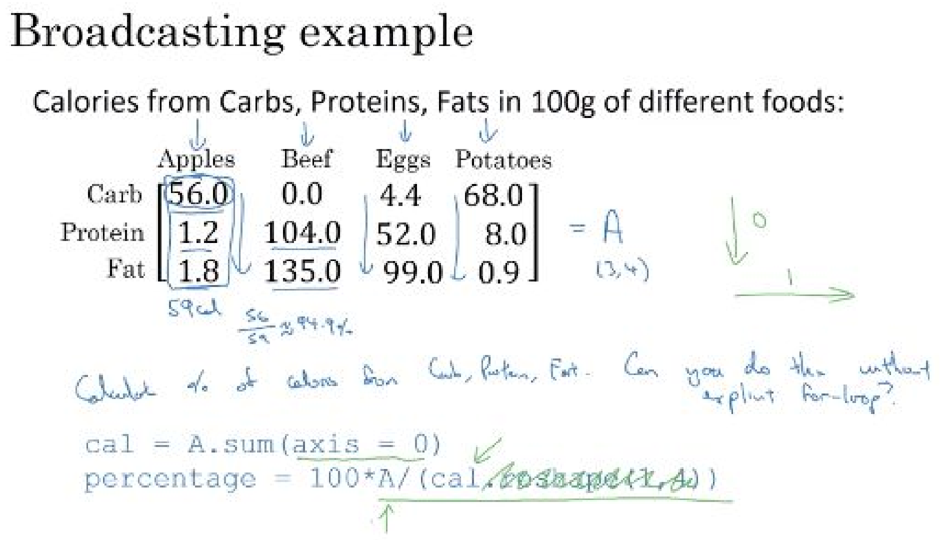
$$\begin{array}{c|cccc} & \text{Apples} & \text{Beef} & \text{Eggs} & \text{Potatoes} \\ \hline \text{Carb} & \boxed{56.0} & 0.0 & 4.4 & 68.0 \\ \text{Protein} & 1.2 & 104.0 & 52.0 & 8.0 \\ \text{Fat} & 1.8 & 135.0 & 99.0 & 0.9 \end{array} = A \quad (3,4)$$

59cal

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

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

- The two lines of python code are shown below. They are
  - $cal \;=\; A.sum(axis = 0)$ adds up the columns (axis=1 adds up the rows)
  - $percentage = 100 * A.reshape(1, 4)$



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

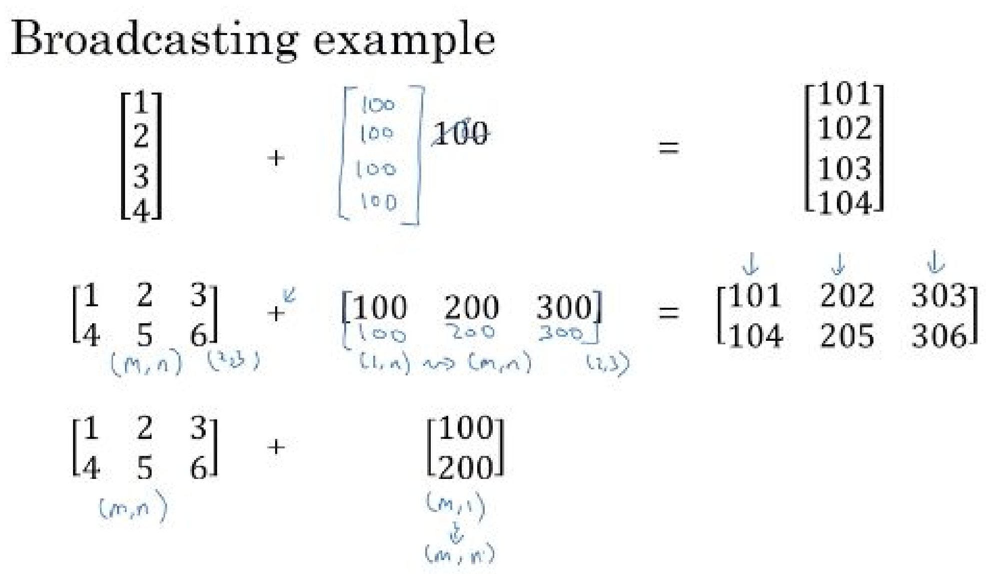
$\frac{56}{59} \approx 94.9\%$

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)~~)
```

Now let's see how we can divide a 3x4 matrix by a 1x4 vector

- Below are more examples on how python will expand a vector to match the other operand's dimension on an operation. In the first example the 100 number is expanded into a 4x1 matrix to match the other operand. In the second example the 3-dimensional vector adds another row to match the other operand. The last example adds two columns to match the dimensions of the other operand.



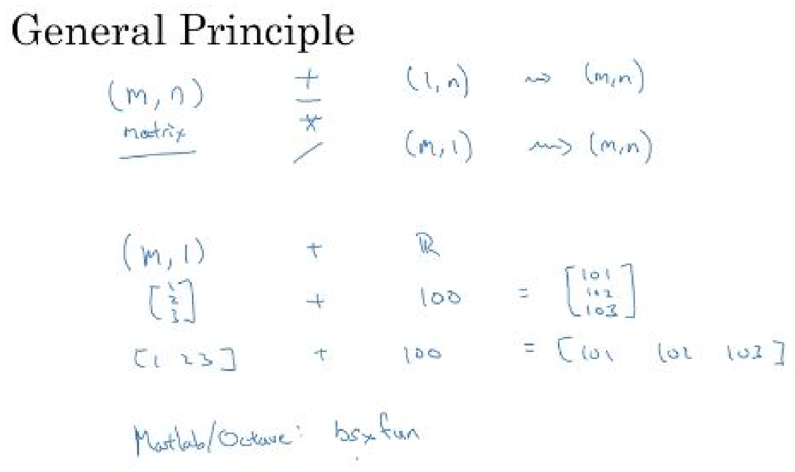
Broadcasting example

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

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

$$\underset{(m,n)}{\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}} + \underset{(m,1) \downarrow (m,n)}{\begin{bmatrix}100\\200\end{bmatrix}}$$

- The general broadcasting principles we are going to use are below. They are:
  - Given an mxn matrix and a 1xn vector: broadcasting will multiply the 1xn vector m times to get a matching mxn matrix
  - Given an mxn matrix and an mx1 vector: broadcasting will multiply the mx1 vector n times to get a matching mxn matrix
  - Given an mx1 vector a a number: broadcasting will multiply the 1x1 number m times to get a matching mx1 vector.



General Principle

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

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

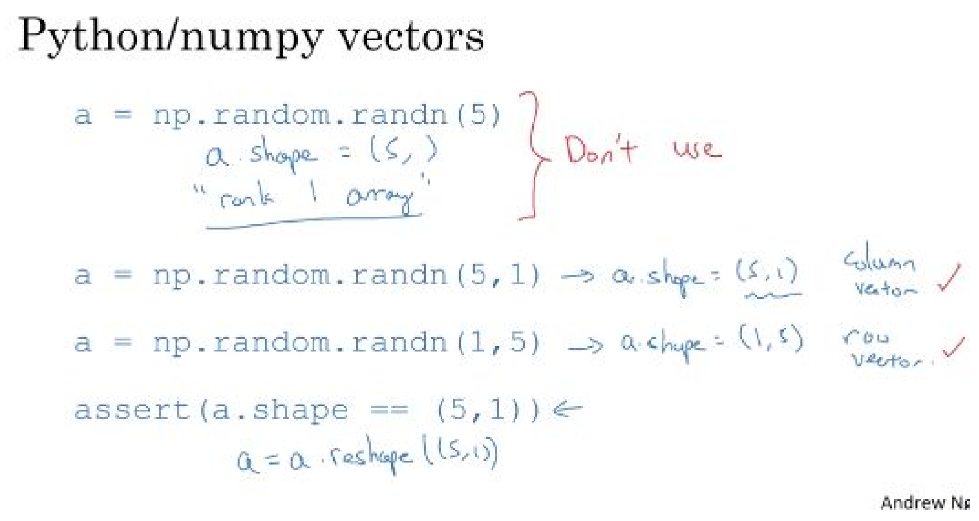
$$\begin{bmatrix}1\\2\\3\end{bmatrix} + 100 = \begin{bmatrix}101\\102\\103\end{bmatrix}$$

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

Matlab/Octave: bsxfun

A note on Python/numpy vectors

- Do not use 'rank 1' arrays in your code because it's a funny data structure that is inconsistent. Rather use explicit column vectors or row vector declarations as show below. Also, if unsure about the shape of a vector, throw an assertion to be sure.



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) column vector ✓

```
a = np.random.randn(1,5)
```

→ a.shape = (1,5) row vector ✓

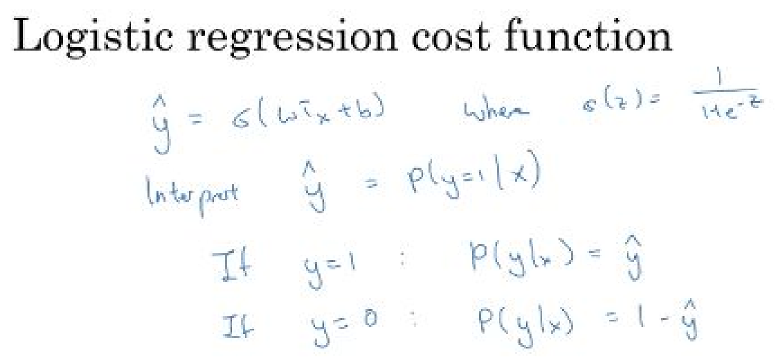
```
assert(a.shape == (5,1))
```

← a = a.reshape((5,1))

Andrew Ng

Explanation of logistic regression cost function (optional)

- We defined the cost function as shown below



Logistic regression cost function

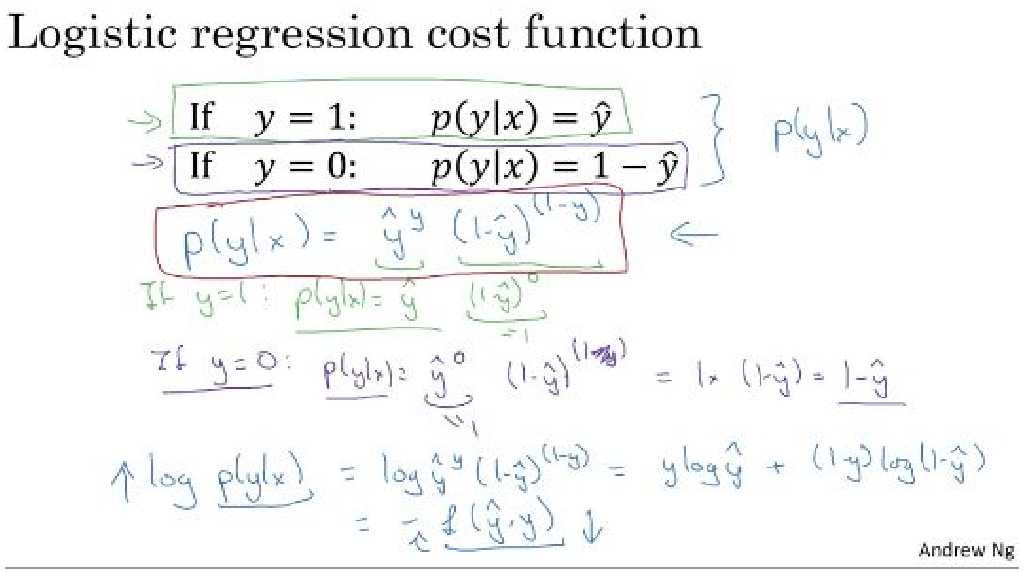
$\hat{y} = \sigma(w^T x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$

Interpret $\hat{y} = p(y=1|x)$

If $y=1$: $p(y|x) = \hat{y}$

If $y=0$: $p(y|x) = 1 - \hat{y}$

- We know that the 2 equations and summarize them into a single equation: $p(y|x) = y^{*y} * (1 - y^{*(1-y)}$ and testing for y=1 and/or y=0 gives us the original two equations.



Logistic regression cost function

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

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

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

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

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

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

Andrew Ng

Learnings from Programming Assignment

- W has to be initialized to a dimension that matches X’s dimension since they are going to be multiplied. Most likely it has to be initialized to the dimensions of your training set’s X.