Lecture Notes: Neural Networks and Deep Learning

Learn from my notes for the Deep Learning course on DeepLearning.ai

Daniel Wu
Dec 13, 2019 · 12 min read ★

Photo by Margarida CSilva on Unsplash

The following are my lecture notes for the first of five courses in the Deep Learning Specialization on DeepLearning.ai:

Week 1 — Introduction

Neural networks are a method to create predictions based on a typically large dataset of inputs with many characteristics called features. The network is made of up units of

until the output node, which presents the outcome.

Week 2 — Binary Classification

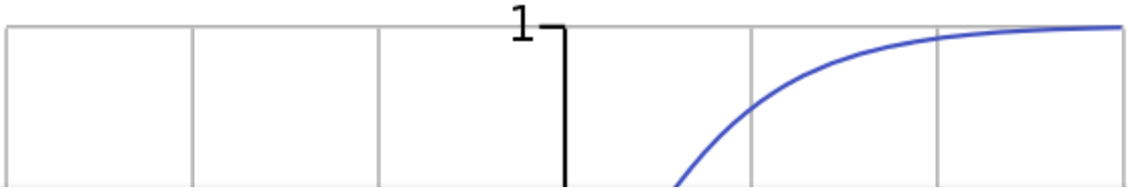
Using logistic regression as an example for neural networks, an image is fed into the network. For example, a color image of a cat is 20 x 20 pixels so it has the image in red, blue, and green channels. That means there are 400 pixels of red, blue, and green each, resulting in 1200 pixels. These 1200 pixels are the number of 'n' features. Input data is unrolled into a column in a matrix of the shape (1200,1). If you have a data set of 100 cat images, then the matrix is (1200,100). Output data in Y matrix is of shape (1200,1) since a logistic regression outputs 1 or 0.

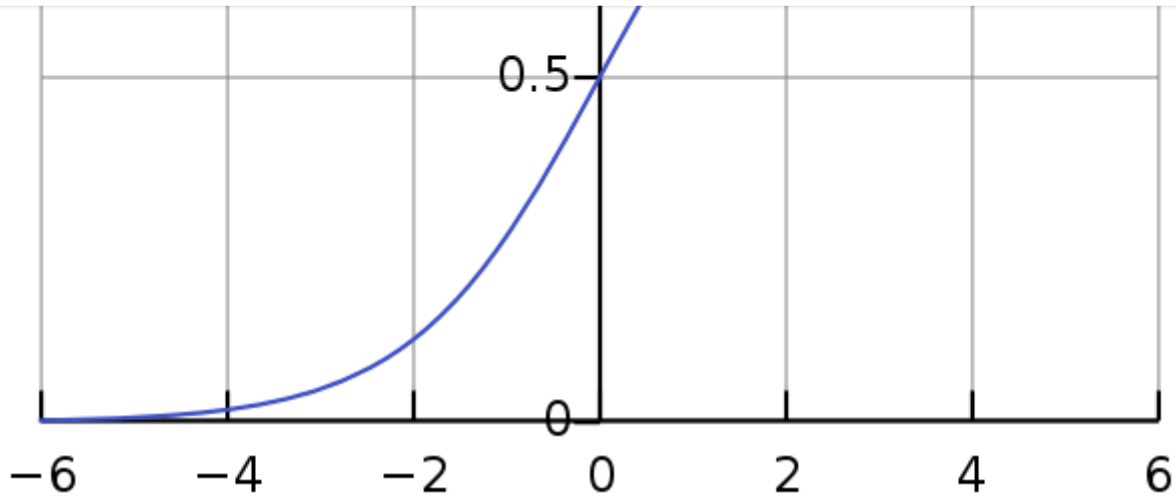
Notations:

- Single training example pair is lower case (x,y)
- For a specific training example 'm' is { (x(i),y(i) … x(m),y(m) }
- Matrix X has x(i) to x(m) stacked in columns
- X shape is (n, m)
- Y shape is (1,m)
- Since I'm not using LaTex or similar mathematical formatting, (i) part of x(i) or any other variable refers to a superscript of i-th example.

Unlike linear regression, a logistic regression applies a sigmoid function so your output is binary (e.g., 0/1). We're training the parameters for 'w' (weight) and 'b' (bias) so the output 'y' best estimated to be 1. We're learning to get the predicted output y-hat(i) approximate as close as possible to actual output y(i).

- z is a function = wTx + b where T is represents the transpose of 'w' matrix
- sigmoid function of z = 1 / (1 + e^-z)





Sigmoid function outputs values between 0 and 1 (Source: wikipedia)

The loss (error) function measures how well our predicted y-hat(i) compares with actual y(i). y(i) notation designates y at the i-th training example. In logistic regression, squared error doesn't do well with gradient descent so the error function L(y-hat,y) = (y-hat — y)^2 / 2. The output is convex and easier to optimize for gradient descent.

$$\text{Loss function} = -(y * \log(\text{y-hat}) + (1-y) * \log(1 - \text{y-hat}))$$

When y = 1, then the loss function L(y-hat,y) is the first part: -log(y-hat) so that large y-hat also means large log(y-hat).

When y = 0, then the loss function L(y-hat,y) is the latter part:-(log (1-y-hat)) so that large log(1-y-hat) means small y-hat.

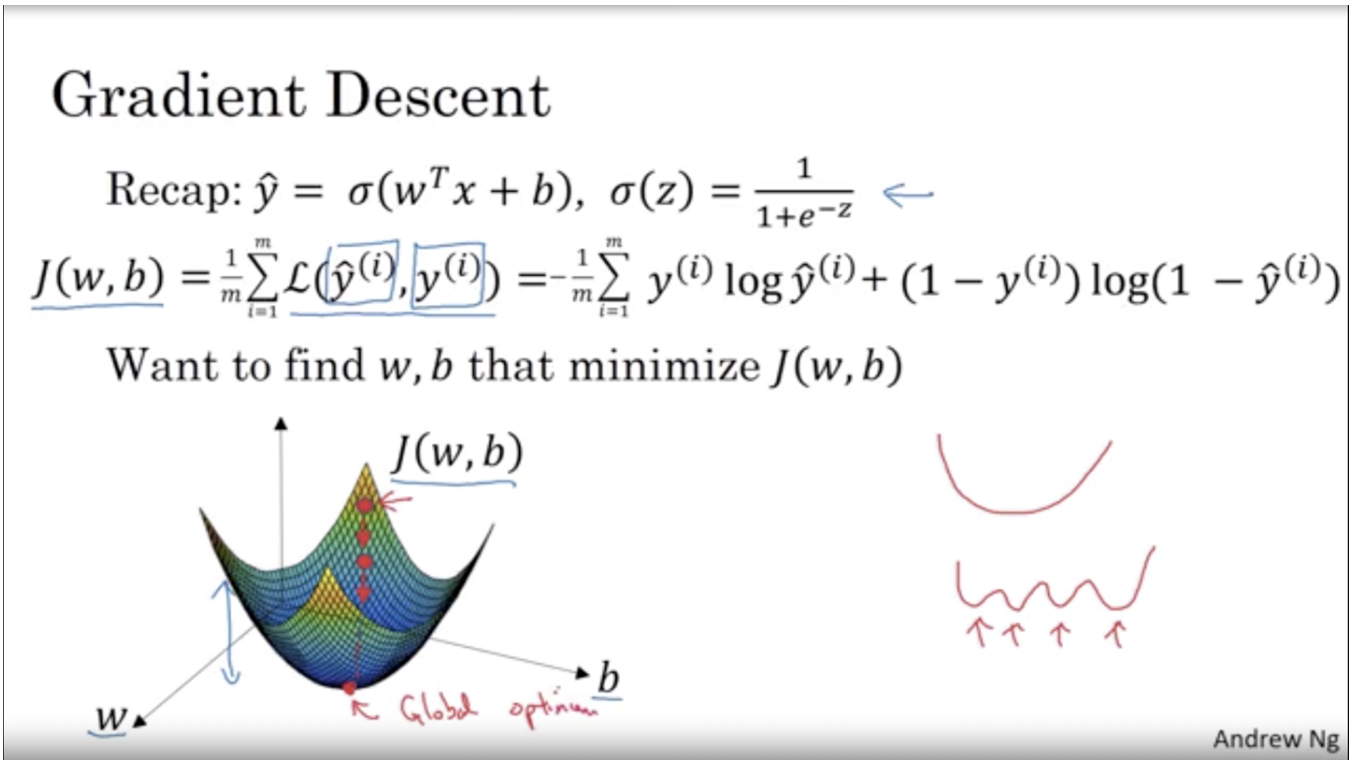
Cost Function:

Since above is defining a loss for a single training example, the cost function is the cost of your parameters based on the loss functions from all the training examples (i to m) averaged by m:

$$J(w,b) = (1/m) * \text{sum}(\ L(\text{y-hat}(i),y(i))\ \text{from i=1 to m}$$

log (1 -y-hat(i))) from i=1 to m

The Gradient Descent goal is to find w,b that minimize cost function J(w,b). Each gradient descent takes steps until the global optimum is reached.



Gradient Descent (Source: DeepLearning.ai / Andrew Ng)

Gradient descent repeats the following where alpha is the learning rate (controls the size of the step of gradient descent) and dJ(w,b)/dw is the derivative that is the slope and known as “dw” in code. w “weight” is being updated as w and derivative changes. b “bias” is being updated as b and derivative changes.

w = w — alpha * dJ(w,b)/dw

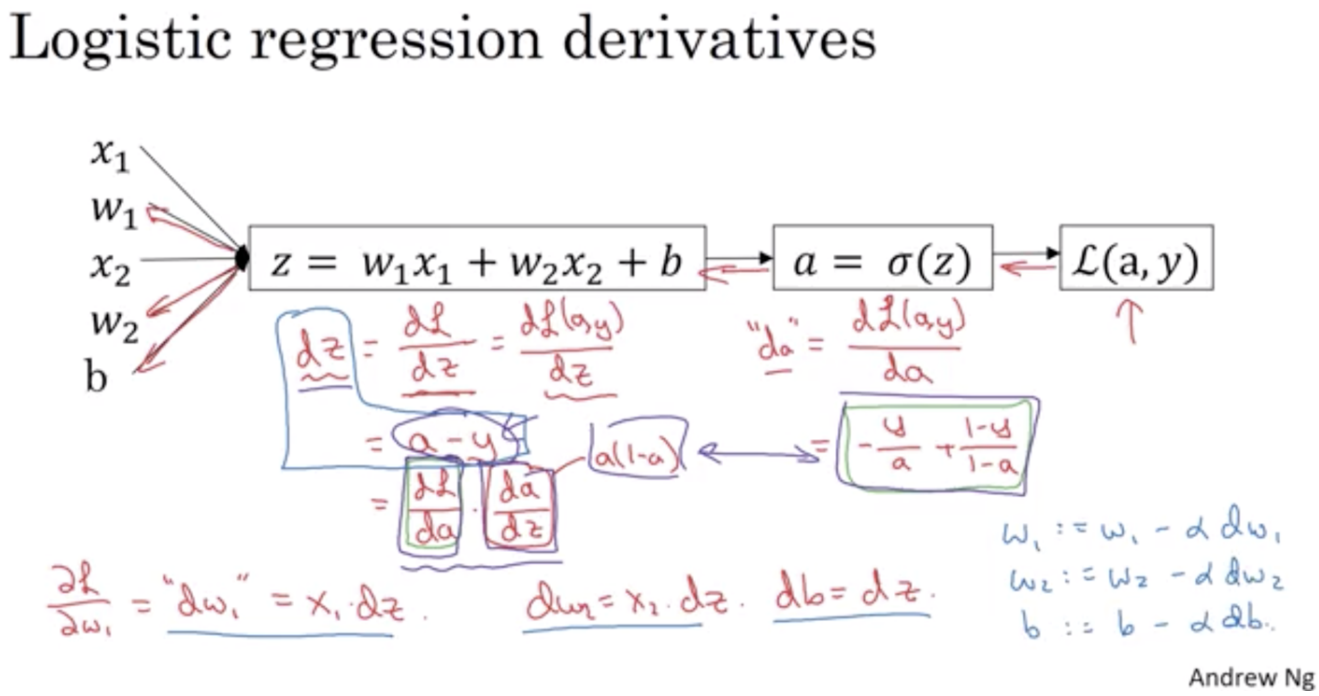
b = b — alpha * dJ(w,b)/db

As the gradient descent steps through the updates, global optimum goes “down hill” until the slope approximates to 0 ideally.

Derivatives with a Computation Graph: This section deals with using a computation graph to calculate derivatives. The exercise plugs in values and finds the ratio by chain

Logistic Regression Gradient Descent:

The following describes the computation graph using logistic regression. Variables x1, w1, x2, w2, and b are inputs into z = w1x1 + w2x2 + b. Z is an input into the sigmoid function, resulting in 'a' which is the same as y-hat. The loss function L is a function of a and y.



Logistic regression derivatives (Source: DeepLearning.ai / Andrew Ng)

The goal of the derivatives is to find the w1, w2, and b that have the minimal loss based on L(a,y). From L(a,y), derivative 'da' can be calculated; from 'a,' derivative dz; and from dz, dw1, dw2, and db. Based on dw1, dw2, and db, the updates to w1, w2, and b can be calculated:

$$da = -(y/a) + (1-y)/(1-a)$$

$$dz = a-y$$

$$dw1 = x1 * dz$$

$$dw2 = x2 * dz$$

Gradient Descent on m Examples:

Recap of cost function J(w,b) which an average of the sum over all the losses:

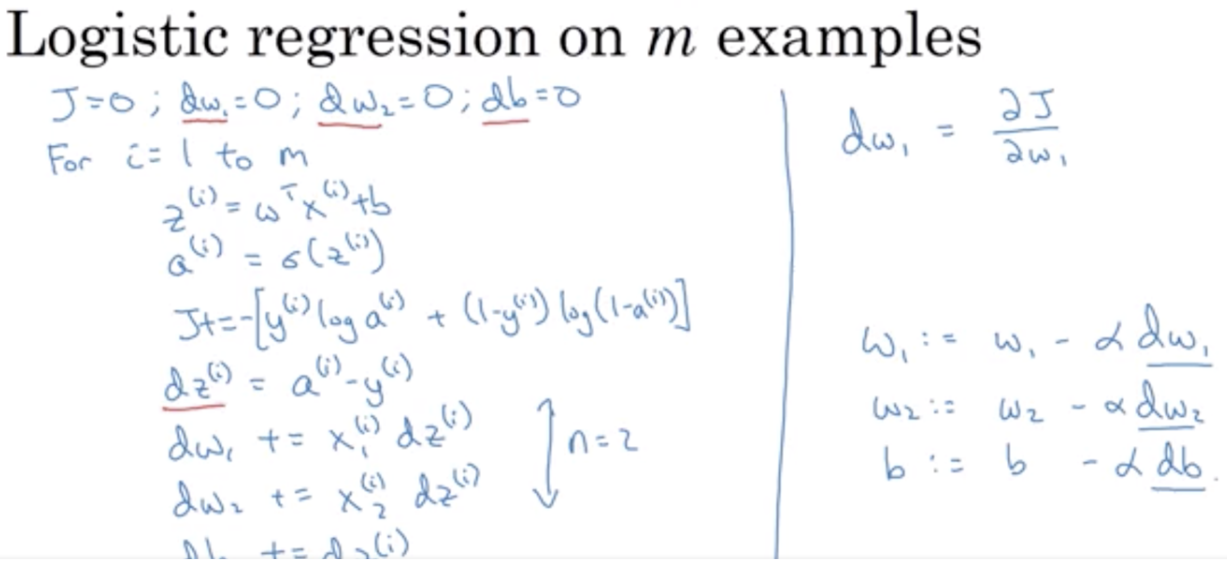
$$J(w,b) = (1/m) * \text{sum}(L(a(i),y(i))) \text{ from } i=1 \text{ to } m \text{ examples.}$$

$$a(i) = \text{y-hat}(i) = \text{sigmoid}(z(i)) = \text{sigmoid}(wT * x(i) + b).$$

The derivative respect to dw1 of cost function J(w,b) is similar in approach and is the average of dw1 across i to m examples:

$$dJ(w,b) / dw1 = (1/m) * \text{sum}(dw1 * L(a(i), y(i)))= dw1(i)-(x(i),y(i))$$

In an algorithm for one step of gradient descent, J, dw1, dw2, db=0 on initialization. In the for loop for i=1 to m, a series of calculation is made to calculate the i-th example of z, a, J, dz, dw1, dw2, db. After the loop, J, dw1, dw2, and db are averaged by m. Once this loop is complete, we get the derivatives of dw1, dw2, and db — not the same as the dw1(i), etc of a single training example. The derivative is used to update the parameters w1, w2, and b.



Logistic regression on m examples (Source: DeepLearning.ai / Andrew Ng)

Note that the above approach shows a single for loop. When there are multiple layers in deep learning, an option is to have a nested for loop within the i=1 to m for loop at the dz(i) to db variables. The better approach with large datasets is to use vectorization which is more efficient.

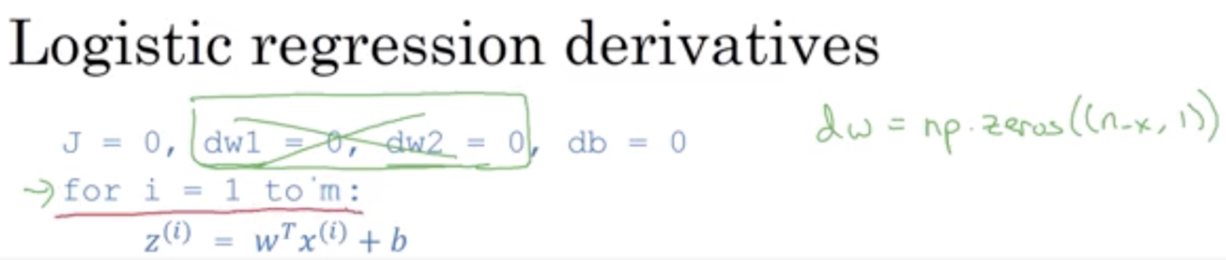
Python and Vectorization:

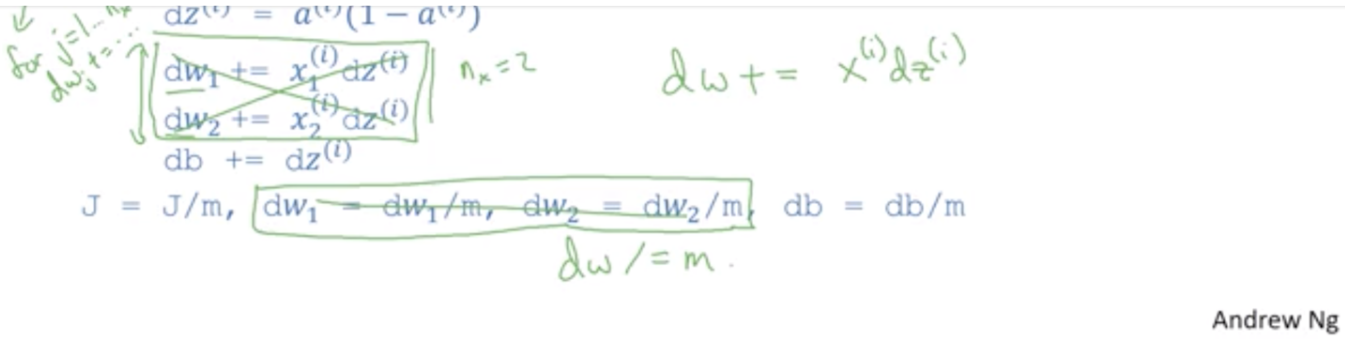
For equation z = wT*x + b where T is transpose of w, the non-vectorized approach looks like the following. In the for loop, we're iterating through i to n examples. Then we're multiplying the i-th 'w' and 'x' together, and updating 'z' with this plus the 'b' bias:

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

Instead, the vectorized approach has w and x in vector column form where both are n of x examples. For example, a 'w' vector of shape (10,1) and 'x' vector of shape (1,5) multiplied together using numpy library's dot() method returns a matrix z of shape (10,5). This eliminates two for-loops and is faster.

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



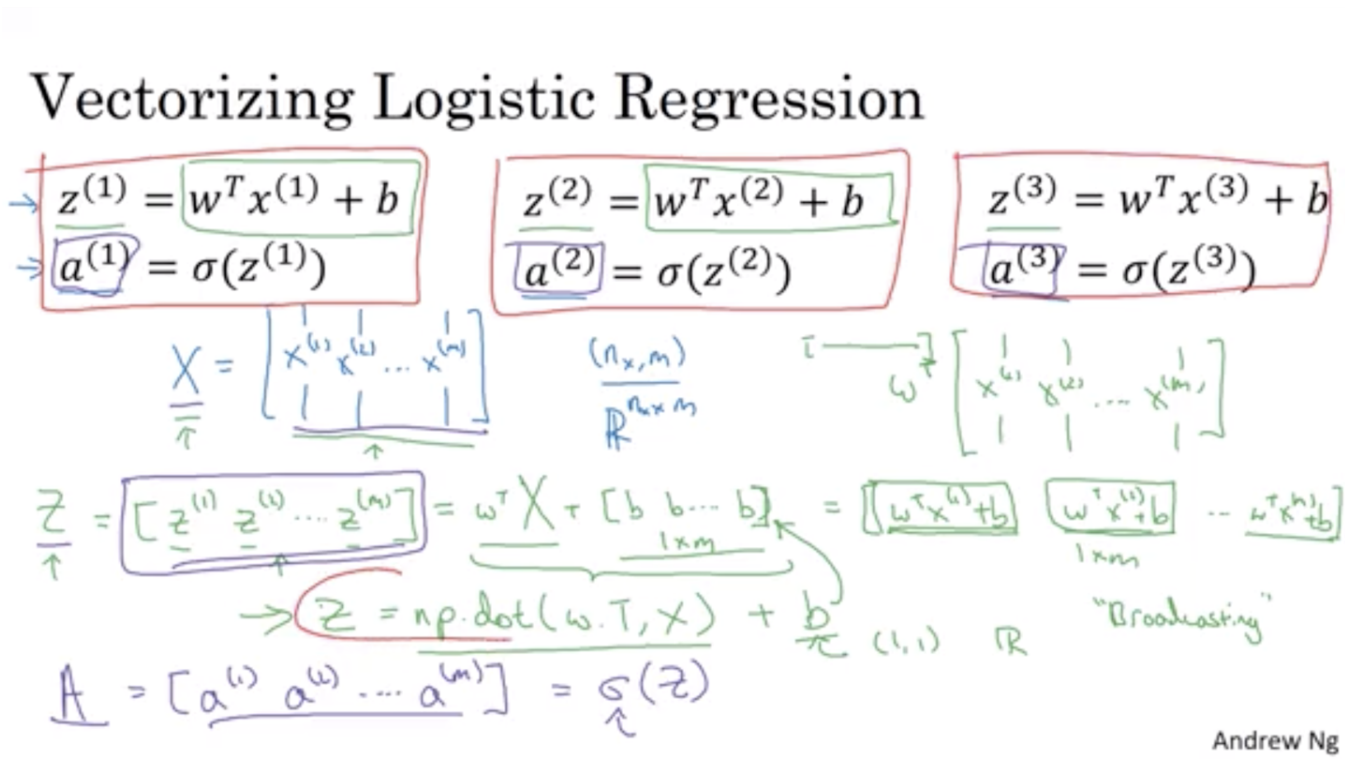


More Vectorization Examples (Source: DeepLearning.ai / Andrew Ng)

Using vectorization / matrix multiplication, the inside for-loop for j=1 to n-x features, dw-j += updates eliminates the dw1 and dw2 updates with a dw += x(i)*dz(i). On the top in green, the respective dw1, dw2 = 0 is replaced with np.zeros(). On the bottom in green, the dw1 = dw1/m and dw2 = dw2/m is simply replaced with dw/=m.

Vectorizing Logistic Regression:

In the forward propagation (left to right equations) of the logistic regression, we stack 'm' training examples of each parameter into matrices. For example x(1…m) is in a X matrix of shape (n-x rows of features, and m columns of each x training examples) or simply (n,m). As result, W, Z, and A are the respective matrices across m training examples. Note: 'b' bias parameter is of shape (1,m).



Vectorizing Logistic Regression's Gradient Output:

In order to remove the outer for-loop along i=1 to m examples, vectorization of the gradients is required. The following set of equations addresses a single iteration of gradient descent for logistic regression (Note: T represents transpose; := notation is simply updating the parameter from the prior value):

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

$$A = sigmoid(Z)$$

$$dZ = A\text{-}Y$$

$$dW = (1/m) * X \,\&\, dZ.T$$

$$db = (1/m) * np.sum(dZ)$$

$$w := w - alpha * dW$$

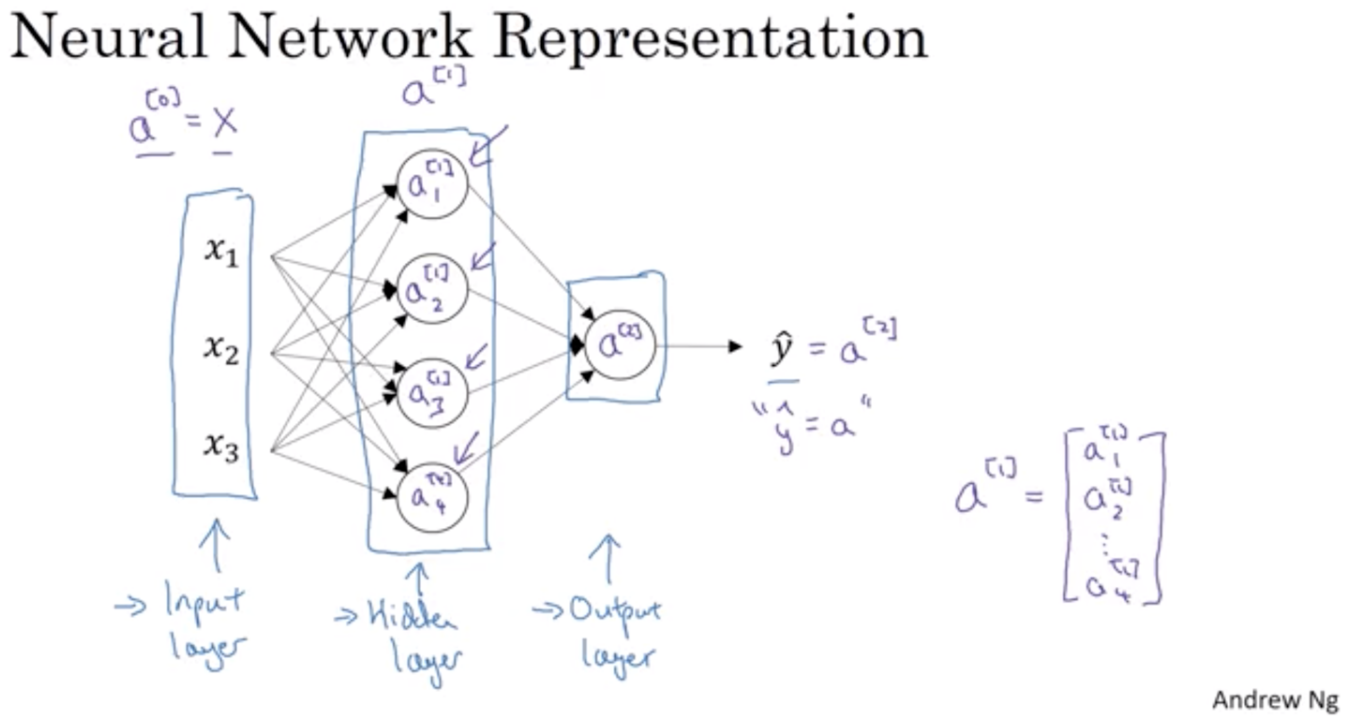
$$b := b - alpha * db$$

The above operations depend on a Python concept called broadcasting. For example, when you have a vector of shape (4,1) and want to add a real number such as 100, then 100 is added to each vector element, resulting in a same shape vector (4,1). Python represents 100 as (4,1). Another example of a matrix (2,3) plus vector (1,3) is interpreted as two rows of (1,3), essentially a (2,3). The resulting matrix is the element-wise operation.

According to Professor Ng, there may be challenging bugs when there are rank 1 arrays (e.g., shape (5,)). Recommendation is to reshape into (n,1) using the shape (e.g., array.shape == (5,1)).

Week 3— Shallow Neural Networks

With this section, a two layer neural network demonstrates how this network works. Notations to be aware of: 1) Previously parentheticals () represented the individual



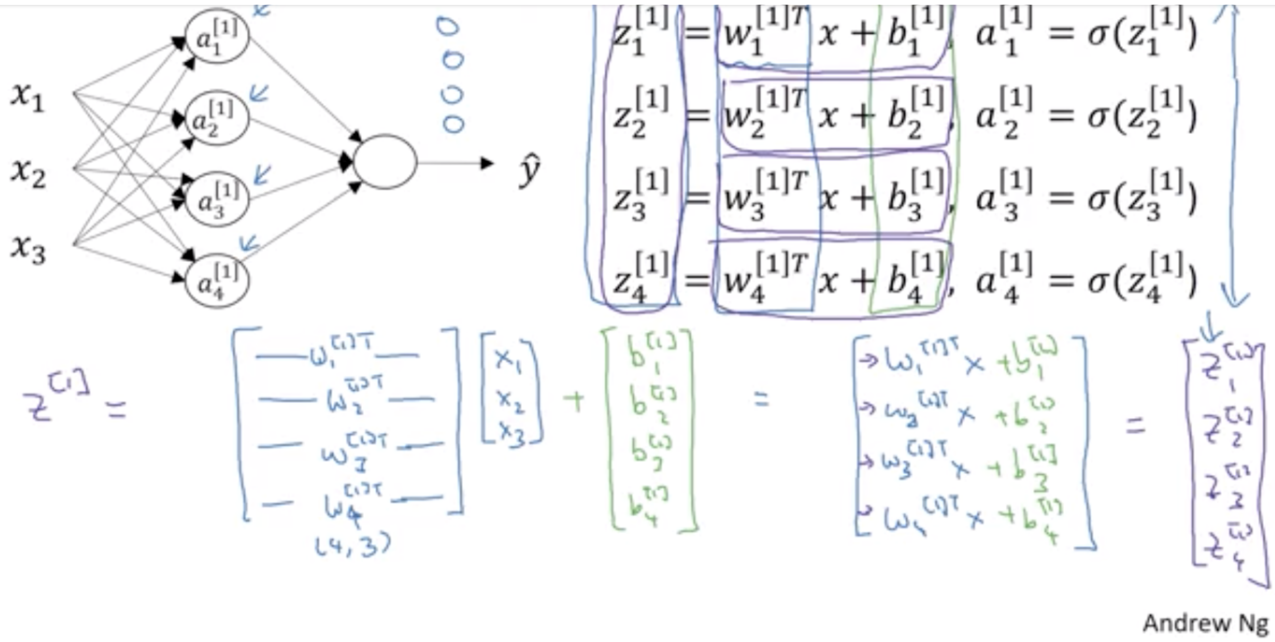
Neural Network Representation (Source: DeepLearning.ai / Andrew Ng)

In the input layer, there are a number of features (e.g., x1, x2, x3 could be a pixel in the RGB layers of a image). The inputs are passed to the hidden layer represented by a superscript [1]. In this example there are 4 nodes each represented as a subscript (number of node) superscript [layer number]. a[1] is a matrix (4,1). In hidden layer 1, a[1] has w, b represented by w[1] and b[1]. Shape of w[1] is (4,3) and b[1] is (4,1). Finally the output is y-hat represented by a[2] and is shape (1,1).

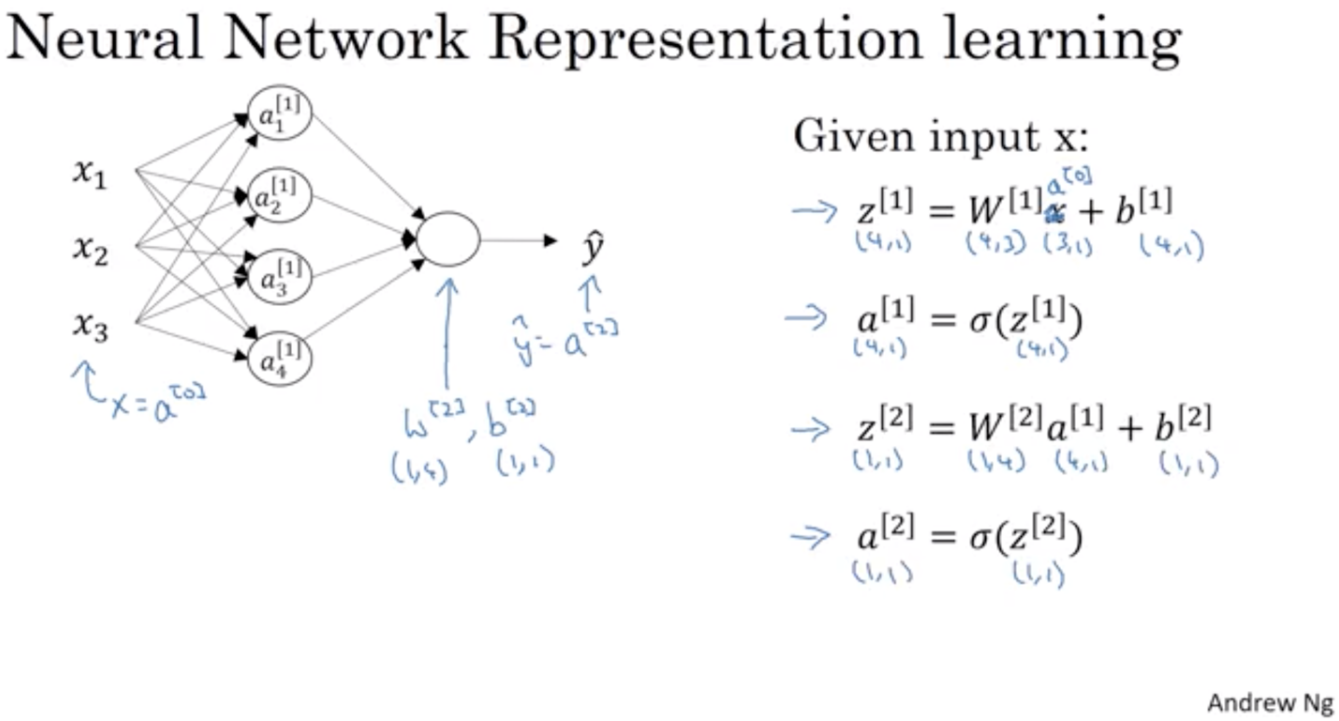
Personally, I keep track of layer matrix shapes with this rule of thumb:

- Input layer is shape (number of features, 1) or (n-x, 1)
- Hidden layer 1 is shape (number of nodes, n-x)
- If there is another hidden layer 2, shape is (number of nodes of hidden layer 2, number of nodes of hidden layer 1).
- Output layer is shape (1, number of nodes of last hidden layer)

Below, the notation a-subscript 1, superscript[1] means the first node of the first layer. The following shows what is behind each activation node 'a' and visualizing the shapes of the matrices:

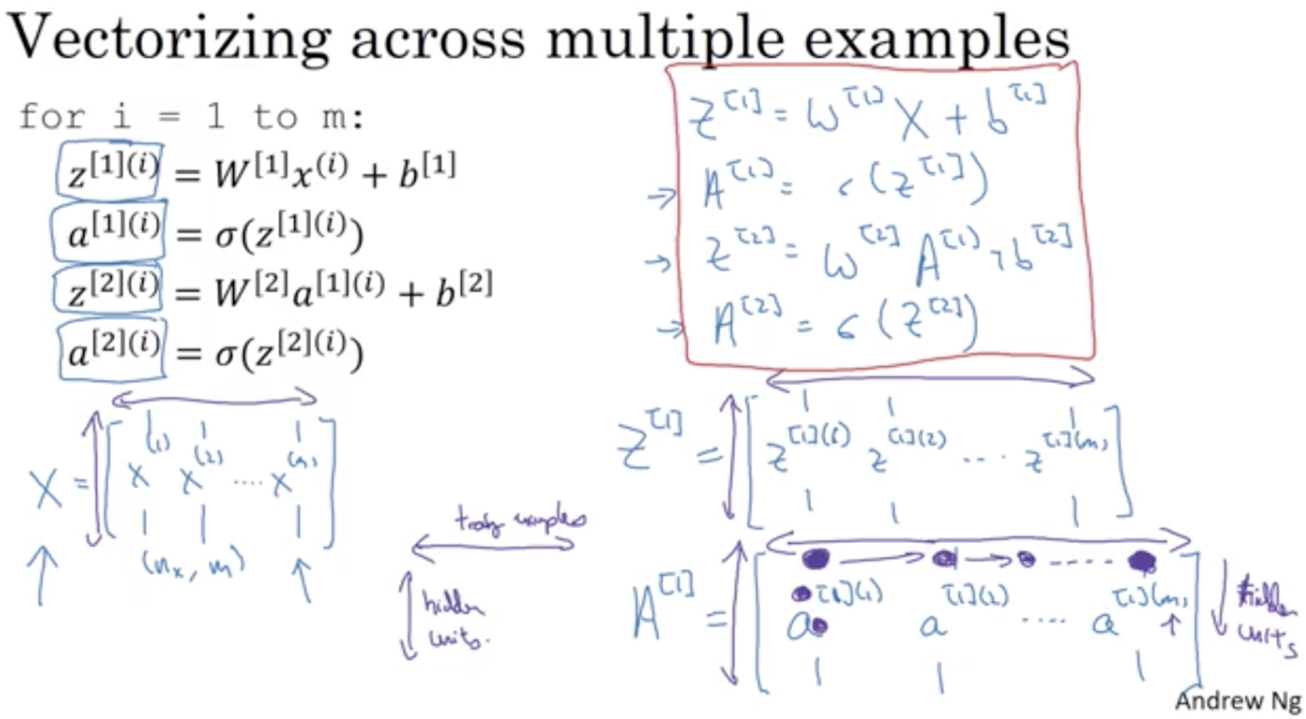


Neural Network Representation (Source: DeepLearning.ai / Andrew Ng)



(Source: DeepLearning.ai / Andrew Ng)

A lot of the debugging in neural networks is ensuring that the shapes of the parameters in the matrices meet expectations. Shape of input, hidden, and output layers are (3,1), (4,3), and (1,4) respectively. Shape of z[1] is (4,1) is because of the W[1] * x is (4,3) multiplied by (3,1), resulting in (4,1). Then, b[1] (4,1) is added by to the previous.



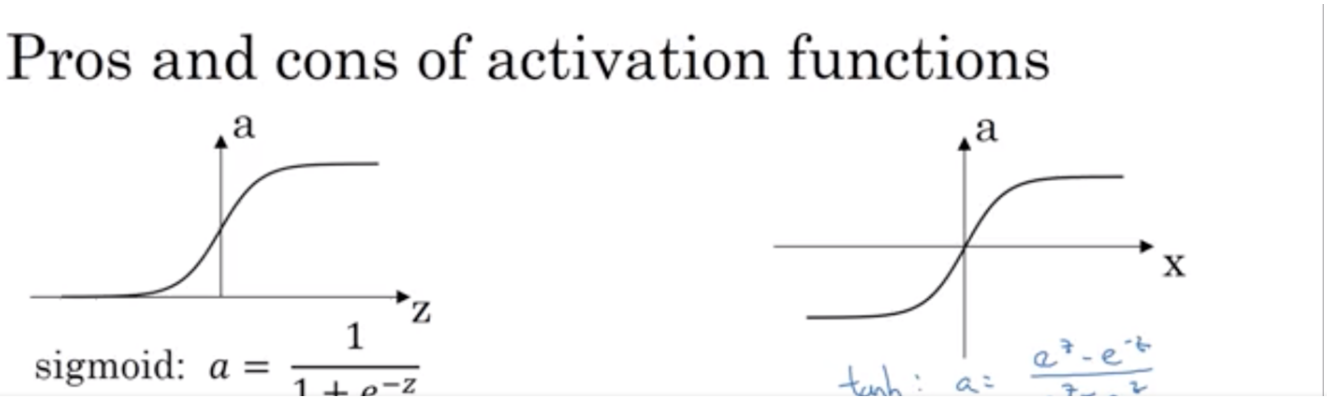
(Source: DeepLearning.ai / Andrew Ng)

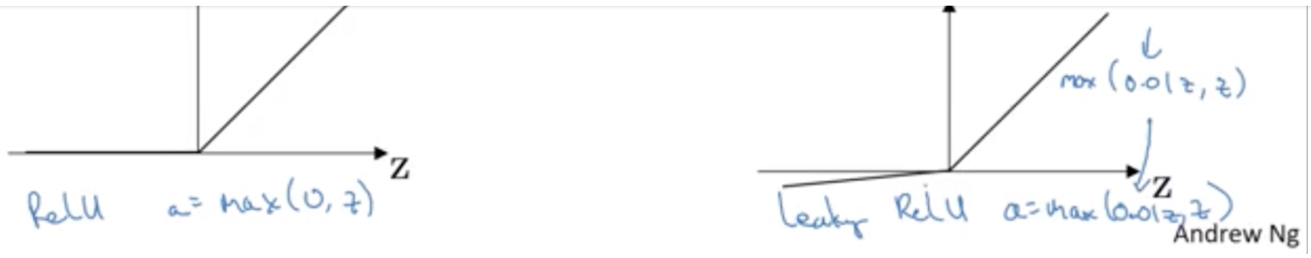
Activation Functions:

When z is very large or very small, the slope of the sigmoid or tan-h function is zero, and can slow down gradient descent calculations.

Instead of sigmoid-based activation function in the hidden layers, the tan-h centers the function around zero. Also increasing popular is the rectified linear unit function (ReLU) where a positive value of z results in a slope of 1; a negative z value is slope zero.

Sometimes a hidden layer would use tan-h or ReLU and the output layer would be sigmoid. Different activation functions by layer are possible.





Activation Functions (Source: DeepLearning.ai / Andrew Ng)

Derivatives of Activation Functions:

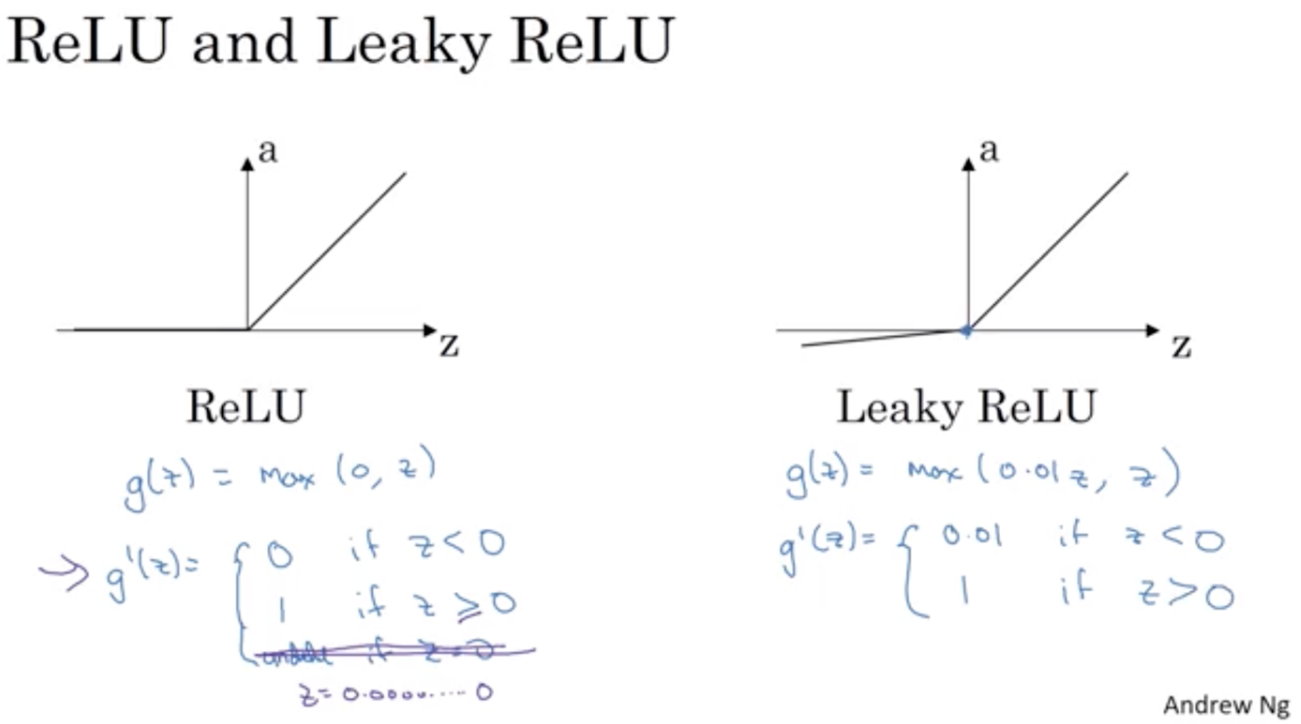
For a sigmoid function g(z), the derivative is the slope of g(z) at z. When z is very large at either positive or negative ends, the slope is zero. The derivative function of activation function with 'a' representing g(z):

$$g'(z) = g(z) * (1-(g(z)) = a * (1-a)$$

For tanh function, the derivative of the activation function:

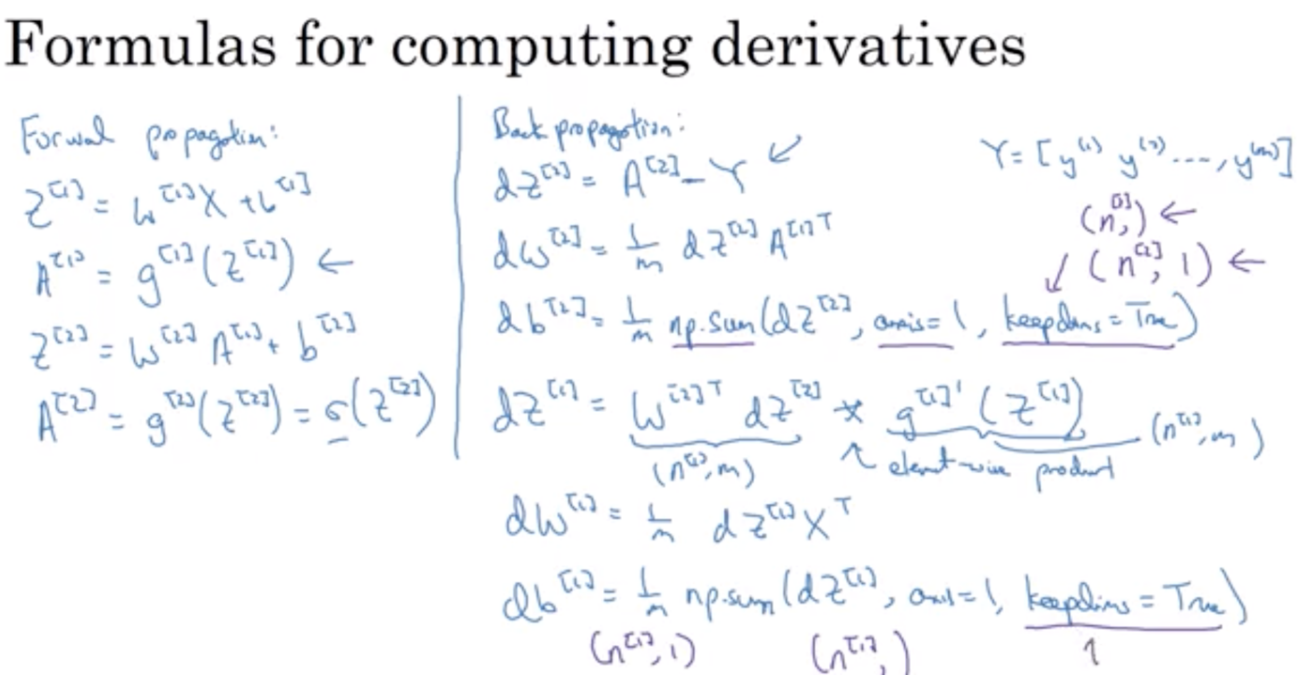
$$g'(z) = 1-(\tanh(z))^2 = 1-a^2.$$

For ReLU and Leaky ReLu functions, the derivatives of the activation functions:



Derivatives of Activation Functions (Source: DeepLearning.ai / Andrew Ng)

forward and back pass being one step of the descent. Initialize parameters randomly and repeat until losses converge.



Gradient descent for neural network (Source: DeepLearning.ai / Andrew Ng)

In a simple neural network with two layers, the forward propagation equations for Z[1] is calculated based on W[1] and b[1]. A[1] is the sigmoid of the Z[1]. Z[2] is the result of W[2] times the prior A, which is A[1], plus b[2]. A[2] is the sigmoid of Z[2]. Z of current layer is always using the prior layer A and so forth.

In the backpropagation, we start with the last hidden layer. dZ[2] is the difference between A[2] and Y, which represents the predicted and actual results. dW[2] = (1/m) * dZ[2] * A[1] transpose. db[2] is the average of the sum of dz[2] where axis=1 and keepdims=True so that the shape of the db[2] matrix will be (n,1) and not (n,), which can result in bugs. dZ[1] is the W[2] transpose * dZ[2] times elementwise multiplication of sigmoid of Z[1]. dZ is shape (n[1],m). Respective formulas are applied to calculate dW[1] and db[1].

Random Initialization

If weights W and b are initialized to zero, each node of the neural network does the same calculation and nothing is learned. Except for b, the solution is to initialize weights randomly.

Beyond a shallow network with a single hidden layer, deep networks can have L number of hidden layers. Essentially, more than single or two hidden layers. The input layer is layer zero. n[L] is the number of nodes/units in layer L.

Getting your matrix dimensions right

The key part of making neural network code work properly is ensuring that we have the right shapes. In an individual training example of z[1] = w[1]* x + b[1], these are the following shapes:

z: (number of nodes for current layer , 1)
w: (number of nodes for current layer , number of input nodes from previous layer)
x: (number of input nodes from previous layer , 1)
b: (number of nodes for current layer , 1)

Shape of w = dw and b = db

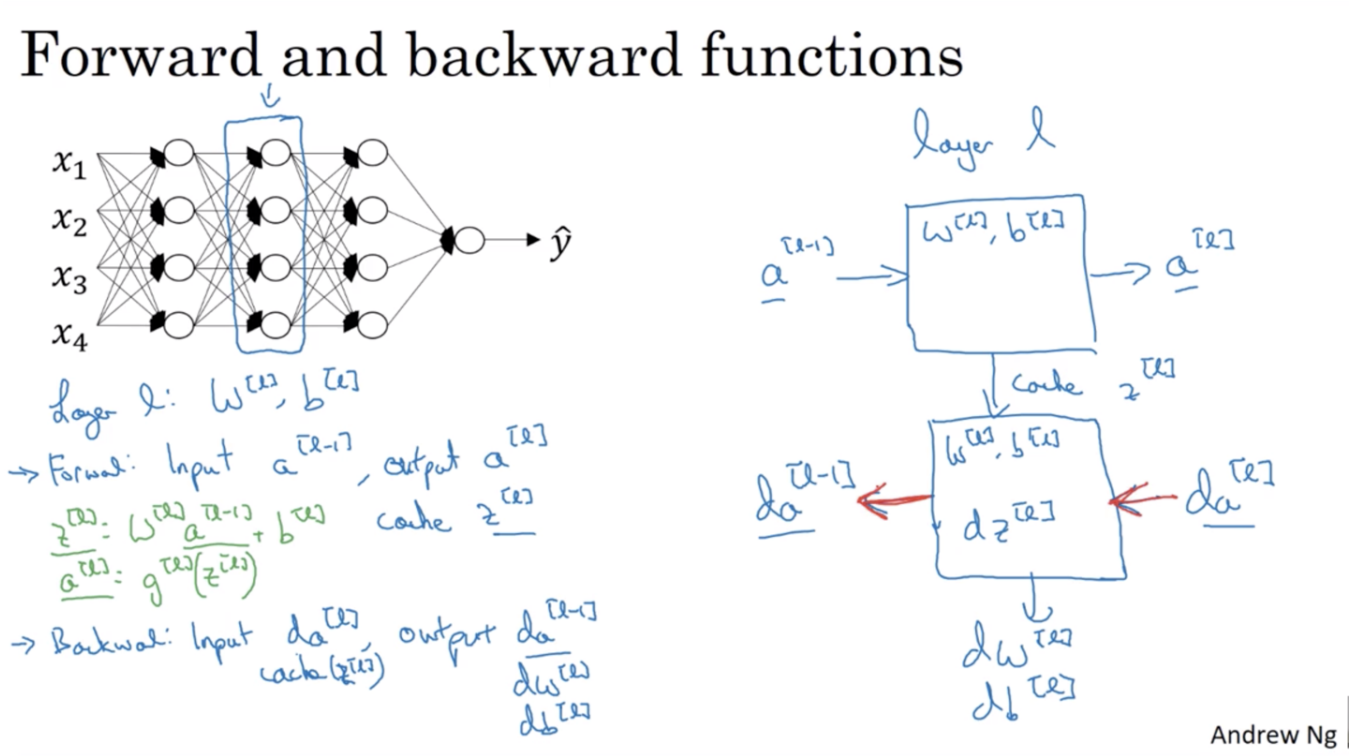
In the vectorized across all training examples for Z = (W*X + B) or (W*A + B), these are the shapes of the matrices:

Z: (nodes current layer , m training examples)
W: (nodes current layer , nodes previous layer)
X or A: (nodes previous layer , m training examples)
B: (nodes current t layer , m training examples)

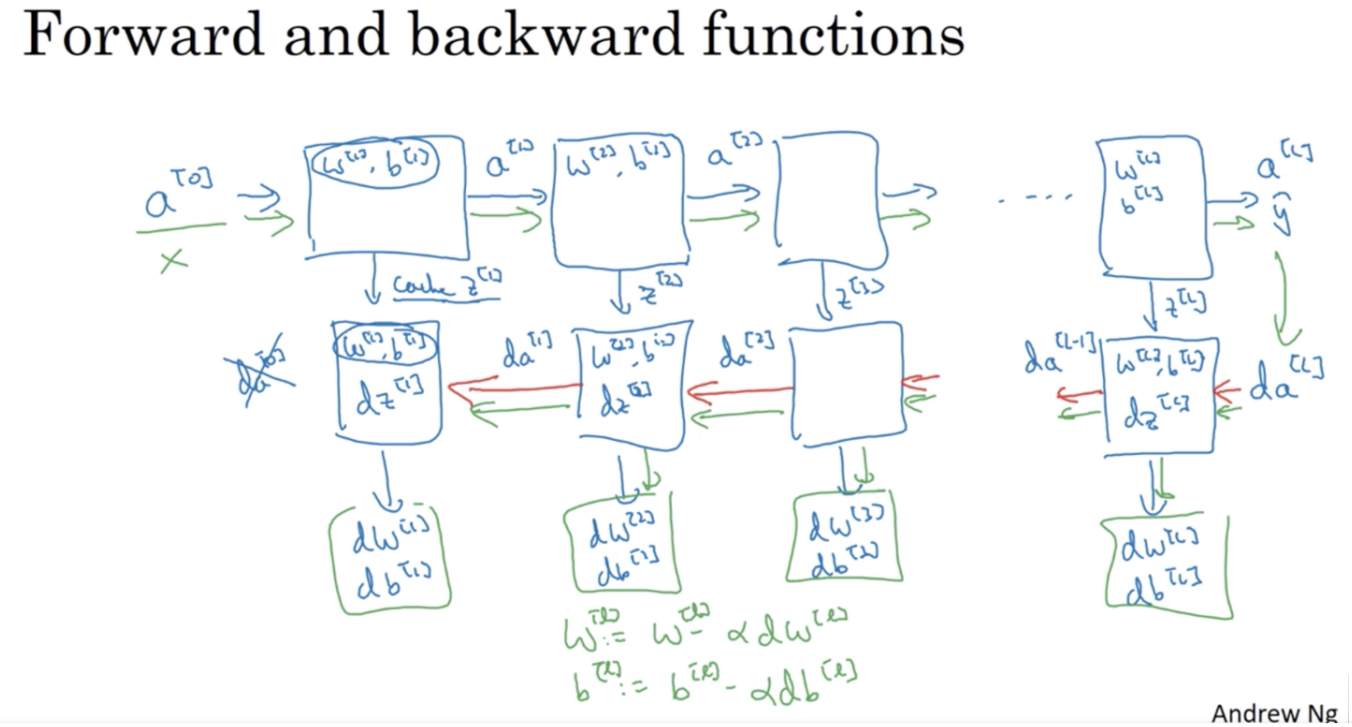
Z = dZ shape and likewise for other derivatives

Building blocks of deep neural networks

propagation, the reverse happens where the derivative of y-hat results in da[l] input to output da[l-1] and parameters dw and db, which are cached like z in the forward propagation. The w and b are updated by the learning rate multipled by the derivatives. All this accounts for one pass of gradient descent to calculate the optimal w and b. The cache is used to keep the parameters for accessibility.

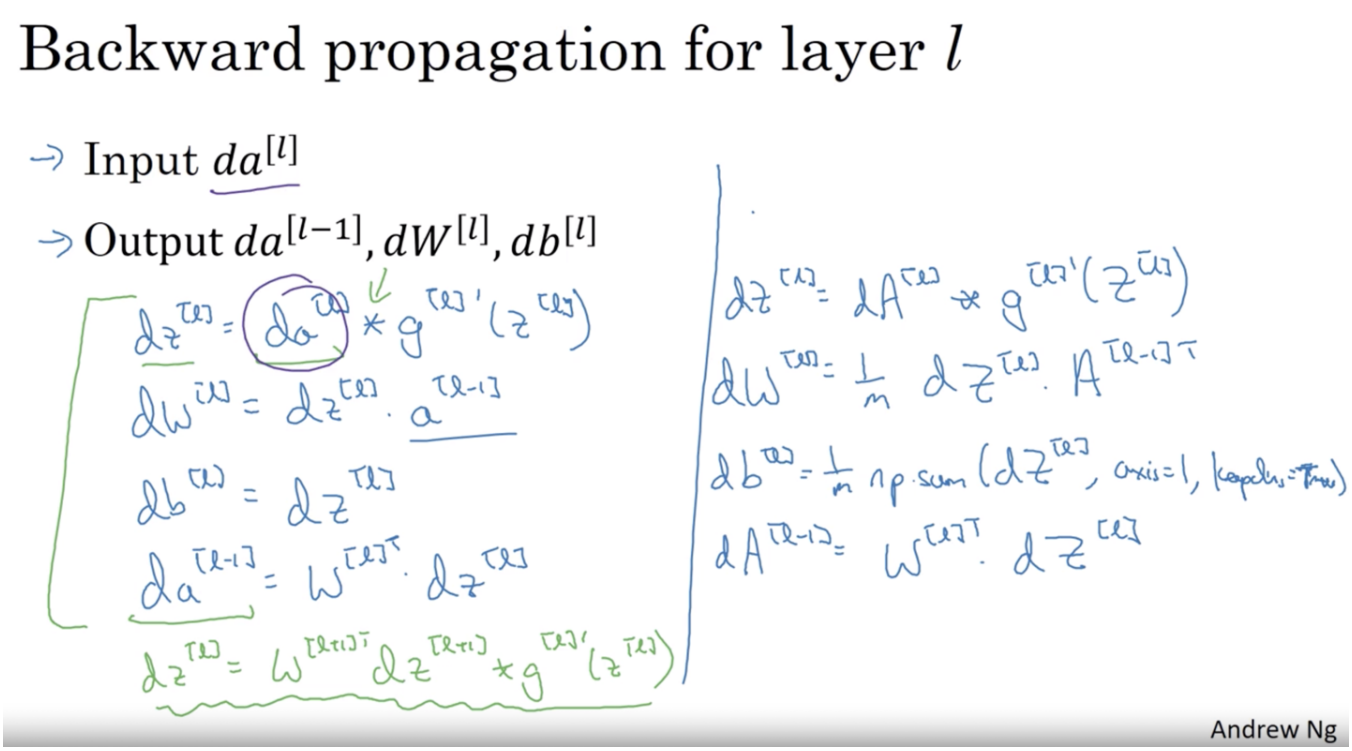


Building blocks of deep neural networks (Source: DeepLearning.ai / Andrew Ng)



Forward and backward propagation

The following shows the calculations for backward propagation using vectorization:



(Source: DeepLearning.ai / Andrew Ng)

. . .

Learn together with me and follow my blog as I share more about machine learning, data science, and product management.

Machine Learning Data Science Deep Learning Neural Networks