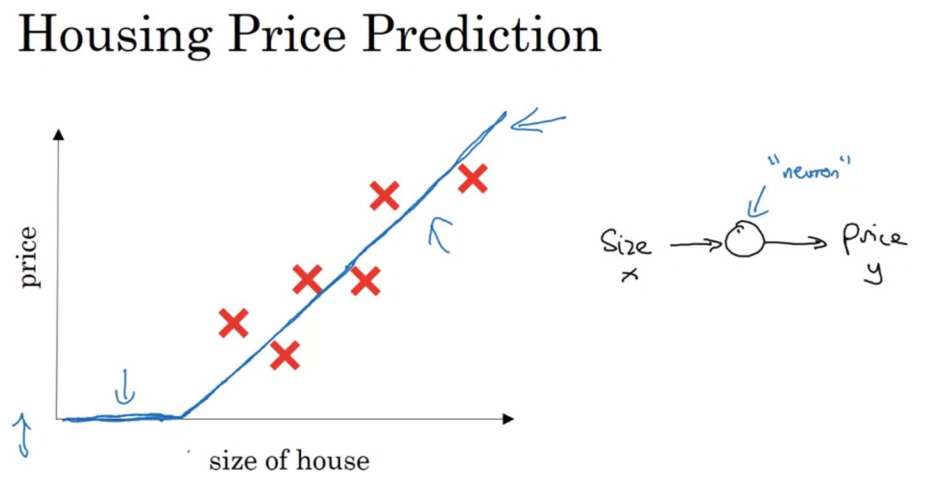
COURSE 1: NEURAL NETWORKS AND DEEP LEARNING

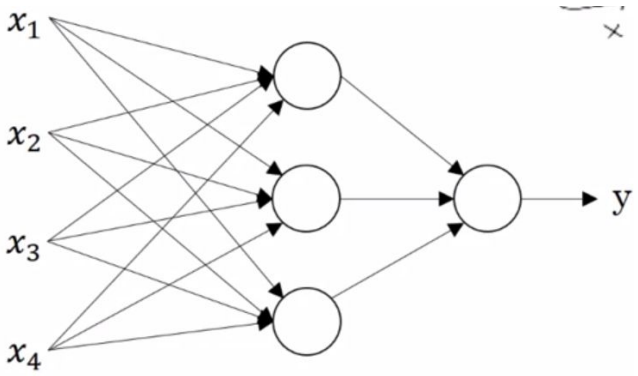
Introduction to Deep Learning

What is a neural network?

Neural networks work like neurons in a brain. If we are dealing with a certain problem with different inputs, we can train the neuron with these inputs to give us an output to a generic input. E.g. - In housing price prediction, we have different values of houses sizes, and different prices for each one of them. If I want to find out what would be a reasonable price for a house, based on its size, we can train the neuron to answer this question using the historical prices that we already know.



A neural network can be composed by just one single neuron, or by lots of neurons (in the case of the Housing Price Prediction, we could have different neurons like: size, # of bedrooms, postal code, quality of life…).



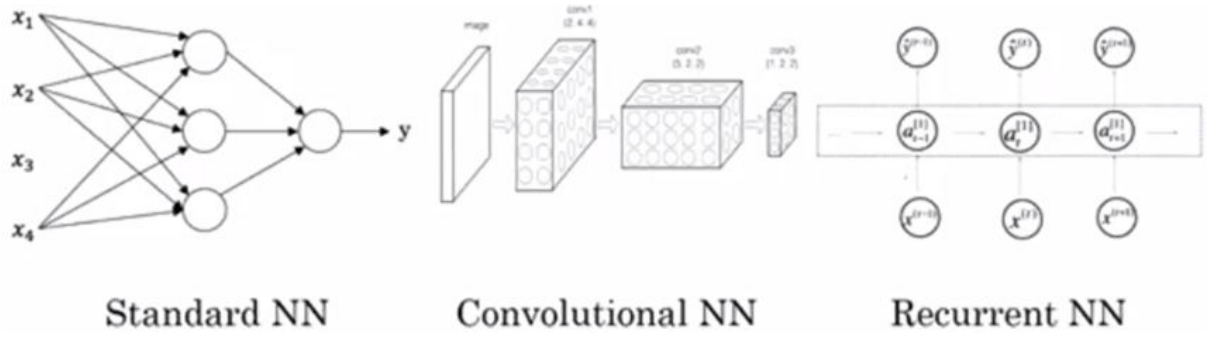
This picture above represents the operation of a neural network. If we give enough data about the problem - different training examples with both x and y - the neural networks are remarkably good at figuring out functions that accurately map from x to y.

Supervised Learning with Neural Networks

In supervised learning, we have some input x, and want to learn a function mapping to some output y. Supervised learning has different applications, like Real Estate, Online advertising, photo tagging, speech recognition, machine translation and autonomous driving, for example. The inputs and outputs for this examples can be seen in the image below:

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

There is an optimal neural network for specific uses. For Real Estate and Online Advertising, we'll often use the standard neural networks. For photo tagging (and computer vision in general), we usually use convolutional neural networks (CNNs). For speech recognition and machine translation, the recurrent neural networks (RNNs) are usually used (because they have sequence data / temporal components). For complex tasks, such as autonomous driving, hybrid and custom neural networks are usually used.



Standard NN Convolutional NN Recurrent NN

Another terms that we can hear about in the supervised learning field, are "Structured Data" and "Unstructured Data". Structure data is the data that is organized in a certain way, with a defined pattern - they're usually seen in databases (like the size of different houses, for example). Unstructured data represents everything else - they have a pattern, but this pattern is not as explicit such as the structured data (images, audio files and text are examples of unstructured data).

Why is Deep Learning taking off?

There is a correlation between the amount of data and the performance you can get with neural networks algorithms. The bigger the amount of data and the size of the neural network, the better the performance usually is. Several years ago, the neural networks were improved a lot due to the increase in the amount of available data and the evolution of computing hardware. Nowadays, there is a big revolution in neural networks performed by the improvement of the algorithms.

The lowercase letter “m” is the notation for the size of the available data.

Neural Networks Basics

Binary Classification

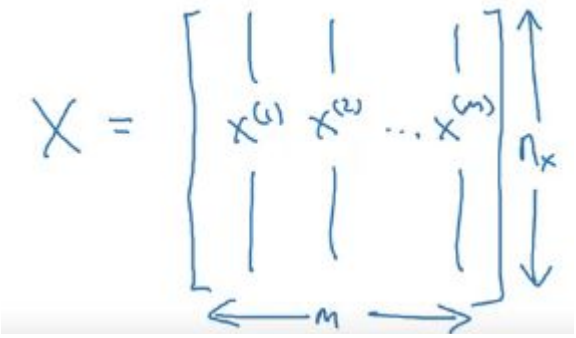
Let’s use an image recognition algorithm for example. If we have a picture of a cat, and want to train a neural network to identify if the image really represents a cat, the output has to be 0 (non cat) or 1 (cat). In this case, the output value is binary (0 or 1). That’s what we call binary classification.

Notation:

$(x, y) \qquad x \in \Re^{n_x}, y \in \{0, 1\}$

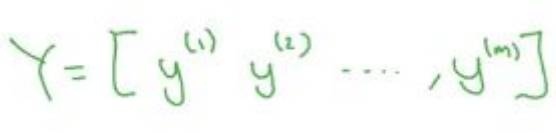
In this notation, x is the input and y is the output of our training examples.

$m \text{ training examples}: \{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)}) \}$



In this image above, the matrix X has all the inputs. Each column of the matrix represents a different training example (with m representing the size of the available data). The variable n_x represents the dimension of the input, so if we have 1000 different data values for each training example, the value of n_x is going to be 1000.

The outputs are represented by a single matrix Y with one row and m columns, as shown in the picture below:



In Python, using the numpy library, we can discover the dimensions of the matrices, using the following command:

Command:

X.shape

Output:

(nx, m)

Command:

Y.shape

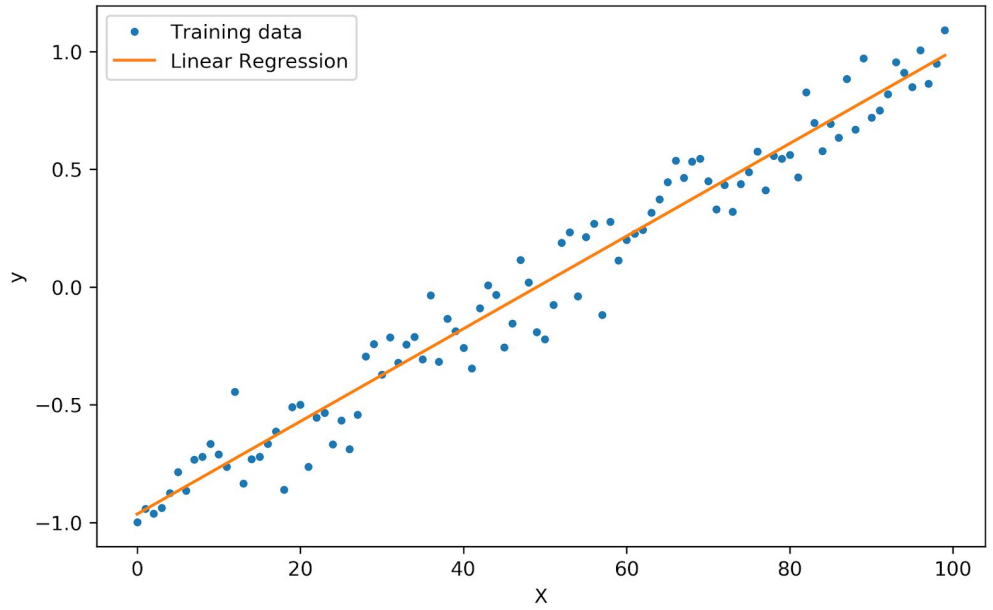
Output:

(1, m)

Logistic Regression

Logistic regression is really helpful when we have to deal with binary outputs. The difference between linear and logistic regression is the function that each one generates. For a given x, $\hat{y}$ is the function that represents the probability of y (the output of the NN) be equals to one. For the parameters $w \in \Re^{n_x}$ and $b \in \Re$ (w is the weight, represented as an n_x dimensional vector, and b is the bias, that is a real number), linear regression generates the following function and plot:

$$\hat{y} = w^T x + b$$



As seen in the plot, it generates a linear function based on the available data.

On the other hand, with the same parameters, logistic regression generates the following function:

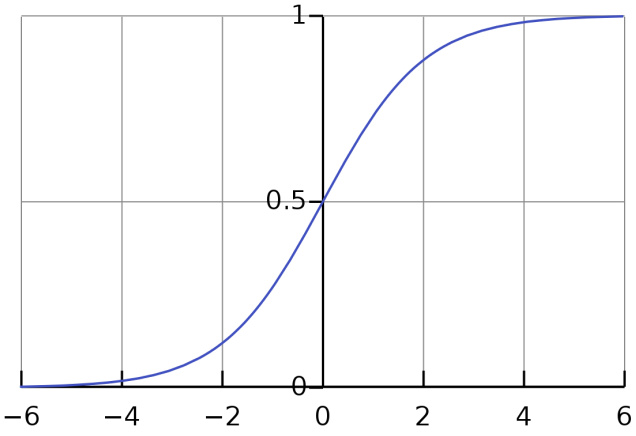
$$\hat{y} = \sigma(w^T x + b)$$

The symbol σ represents the sigmoid function. If we call $z = w^T x + b$, the sigmoid function of z is given by the following expression:

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

Analysing the function, if z is a very large positive number, then the sigmoid function will be approximately close to 0. If z is a very large negative number, the sigmoid function will be close to 0 (1/large number $\simeq$ 0).

The plot of $\hat{y}$ can be seen below:



Logistic Regression cost function

To train the parameters w and b of the logistic regression, we need to define a cost function. The parameters w and b are very important, because if we choose the right values for them, the output function ŷ is going to be very close to the truth value. The notation $^{(i)}$ is usually used to represent the i-th training example (training example number i). Using this notation, we can rewrite the equations from the previous topic:

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

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

Given some training examples $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$, we want $\hat{y}^{(i)} \approx y^{(i)}$ -> the computed output to be close to the real value.

The loss (error) function, represents the difference between the computed value and the expected value. In linear regression, the loss function is defined by the following equation:

$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

But for logistic regression, this function doesn't work very well. So we usually use the following equation instead:

$L(\hat{y}, y) = -(y log\hat{y} + (1-y) log(1-\hat{y}))$

If $y = 1: L(\hat{y}, y) = -log\hat{y}$ -> in this case, we want that the value of $-log\hat{y}$ be as small as possible, so we want a big $\hat{y}$ (in this case, the bigger is ŷ, the smaller is -logŷ).
If $y = 0: L(\hat{y}, y) = -log(1-\hat{y})$ -> in this case, we want that the value of $-log(1-\hat{y})$ be as small as possible, so we want a small $\hat{y}$ (in this case, the smaller is ŷ, the smaller is -log(1-ŷ)).

With the loss function defined, we can now determine the cost function for a logistic regression:

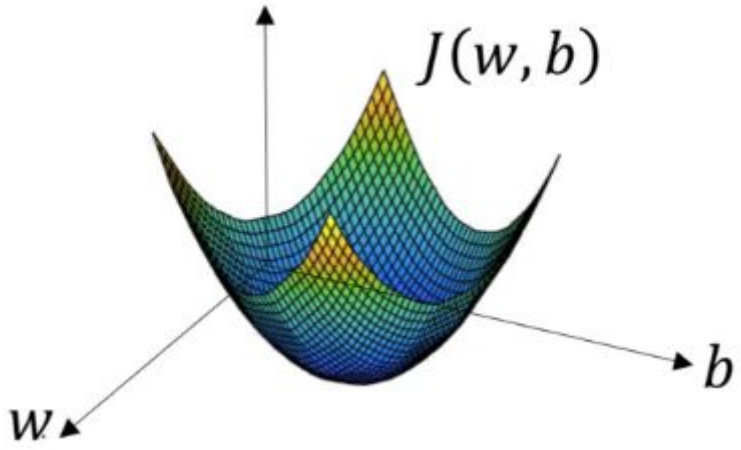
$Cost\ function:$

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

In other words, the loss function computes the error for a single training example, while the cost function is the average of the loss functions of the entire training set.

Gradient descent

The idea of the gradient descent algorithm is to find the parameters w and b that minimize the cost function. The cost function is a convex function, this means that it has only one optimal value (in this case, the optimal value is the minimum value that the function can assume).



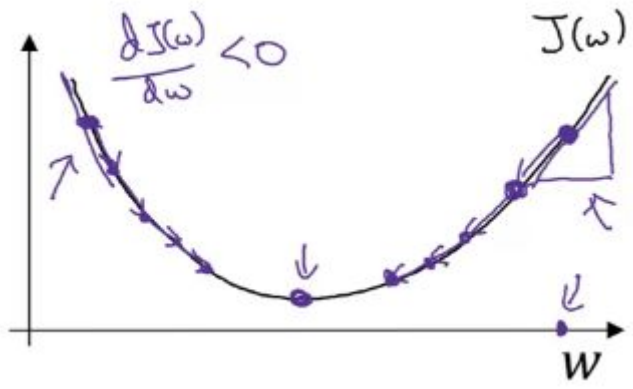
The initial values of w and b are rarely optimal. In this algorithm, each iteration improves the values of w and b, reaching at the end of the execution the optimal values that minimize the cost function. To exemplify the functioning of this algorithm, let's consider a problem with a constant b. In this case, the only variable is w. The algorithm would be like this:

Repeat{

$$w := w - \alpha \frac{dJ(w)}{dw}$$

}

In the equation above, α is the learning rate, that controls how big a step we take on each iteration on gradient descent. The derivative represents the update we want to make on the parameter w. The notation commonly used in programming languages is to call the whole derivative as just "dw" ($\frac{dJ(w)}{dw} \rightarrow dw$). The repeat process can be seen below:



If b variates as well, we could use the following repeat process:

Repeat{

$$w := w - \alpha \frac{dJ(w, b)}{dw}$$

$$b := b - \alpha \frac{dJ(w, b)}{db}$$

}

Derivatives

The derivatives (slopes) examples in the image below are more than sufficient for our applications. Don't worry about decorating them, if you forget any derivative just come here and check its value.

Product Rule: $\frac{d}{dx}(uv) = u'v + uv'$ Quotient Rule: $\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{u'v - uv'}{v^2}$

1. $\frac{d}{dx} c = 0$
2. $\frac{d}{dx} x = 1$
3. $\frac{d}{dx} cx = c$
4. $\frac{d}{dx} x^n = nx^{n-1}$
5. $\frac{d}{dx} e^x = e^x$
6. $\frac{d}{dx} \ln x = \frac{1}{x}$
7. $\frac{d}{dx} a^x = a^x \ln a$
8. $\frac{d}{dx} \log_a x = \frac{1}{x} \cdot \frac{1}{\ln a}$
9. $\frac{d}{dx} \sin x = \cos x$
10. $\frac{d}{dx} \cos x = -\sin x$
11. $\frac{d}{dx} \tan x = \sec^2 x$
12. $\frac{d}{dx} \cot x = -\csc^2 x$
13. $\frac{d}{dx} \sec x = \sec x \tan x$
14. $\frac{d}{dx} \csc x = -\csc x \cot x$
15. $\frac{d}{dx} \arcsin x = \frac{1}{\sqrt{1 - x^2}}$
16. $\frac{d}{dx} \arccos x = \frac{-1}{\sqrt{1 - x^2}}$
17. $\frac{d}{dx} \arctan x = \frac{1}{1 + x^2}$
18. $\frac{d}{dx} \operatorname{arccot} x = \frac{-1}{1 + x^2}$
19. $\frac{d}{dx} \operatorname{arcsec} x = \frac{1}{|x|\sqrt{x^2 - 1}}$
20. $\frac{d}{dx} \operatorname{arccsc} x = \frac{-1}{|x|\sqrt{x^2 - 1}}$

Chain Rule

If f and g are both differentiable and $F(x)$ is the composite function defined by $F(x) = f(g(x))$ then F is differentiable and F' is given by the product

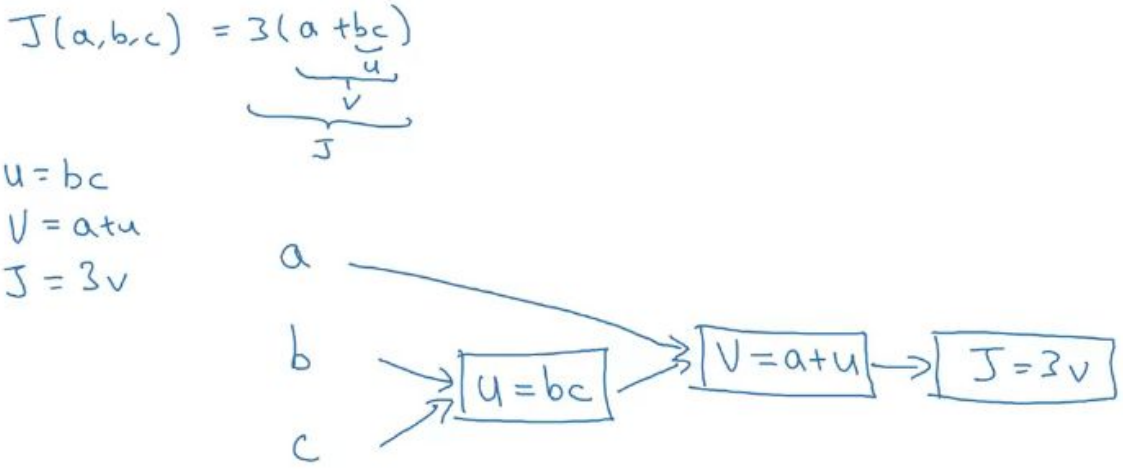
$$F'(x) = f'(g(x))\, g'(x)$$

Differentiate outer function

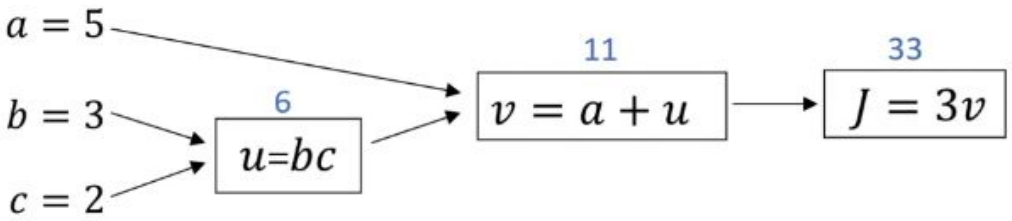
Differentiate inner function

Computation Graph and Derivatives with a Computation Graph

Neural networks are organized in terms of a forward pass (forward propagation step), in which we compute the output of the neural network, followed by a backward pass (back propagation step), which we use to compute gradients or derivatives. The computation graph explains why the neural network is organized this way. The following image represents a simple example of a computation graph:



We use substitutions to simplify the original function, creating little “rectangles” that have the substitutions inside them. If we want to calculate the output of the neural network, we just have to go from the left to the right of the graph. If we want to calculate derivatives or gradients, we might need to go from the right to the left. An example of computing derivatives can be seen in the image below:



In this case, after we compute the output (to compute the output we used the forward pass), we might want to calculate the derivative of J with respect to a. First of all, lets calculate the derivative of J with respect to v:

$\frac{dJ}{dv} = 3$

And then evaluate the value of the derivative of v with respect to a:

$\frac{dv}{da} = 1$

With these derivatives calculated, we can now relate J with a (if a changes its value, how much it will affect the value of J?). We will have, like we see in calculus courses, a chain rule, given by the following representation:

$\frac{dJ}{dv} * \frac{dv}{da} = 3 * 1 = 3 = \frac{dJ}{da}$

In conclusion, to calculate the derivative of J with respect to other variables, we might need to do a backward propagation step to be able to compute the value of the derivative.

In the example above, J is called the final output variable. In a lot of problems, we want to calculate the derivative of J with respect to other variables. The derivative will be like this:

$$\frac{dFinalOutputVar}{dvar}$$

Instead of trying to use a huge name to represent this derivative in the codes, we use the notation already seen in the gradient descent notes, and call this whole derivative “dvar”. So:

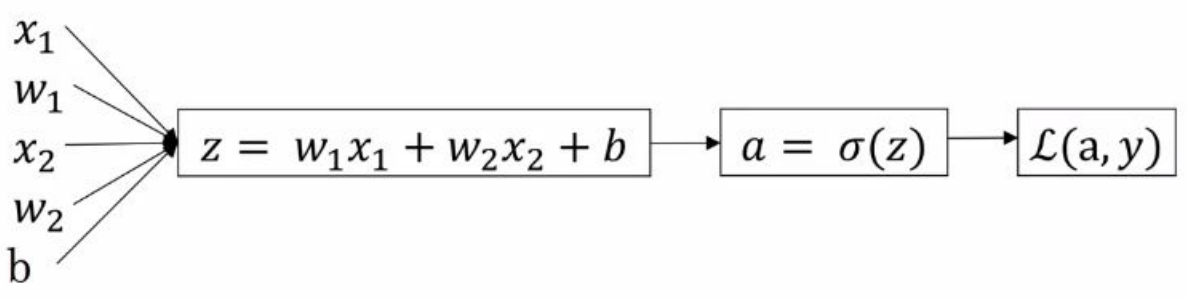
$\frac{dJ}{da} = da \quad ; \quad \frac{dJ}{dv} = dv$

Logistic Regression Gradient Descent

Let’s do an example of derivative calculation using gradient descent. For this example, let’s say we have only two features, x1 and x2.

$$z = w^T x + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L}(a, y) = -(y \log(a) + (1 - y) \log(1 - a))$$

In order to compute z we’ll need to input w1, w2 and b in addition to x1 and x2. We’ll have the following computation graph:



Let's compute the derivative of the loss function, with respect to all the other variables. We first go backwards to the expression $a = \sigma(z)$.

$\frac{dL(a,y)}{da} = \frac{-y}{a} + \frac{(1-y)}{(1-a)}$ (in python = "da")

Then, we go backwards to the expression of z:

$\frac{dL(a,y)}{dz} = \frac{dL(a,y)}{da} * \frac{da}{dz} = \left(\frac{-y}{a} + \frac{(1-y)}{(1-a)}\right) * (a * (1-a)) = a - y$ (= "dz")

Observation: the derivative of a with respect to z involves the derivative of the sigmoid function, that can be seen below:

$$Sigmoid(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{dx}S(x) = \frac{d}{dx}\frac{1}{1+e^{-x}}$$

Using the quotient rule to take the derivative:

$$\frac{d}{dx}S(x) = \frac{e^{-x}}{(1+e^{-x})^2}$$

At this point, we're done, but it doesn't look friendly and computationally useful. So we use a simple technique: adding and subtracting the same thing (which changes nothing) to create a more useful representation. In this case we add and subtract 1 in the numerator.

$$\frac{d}{dx}S(x) = \frac{1-1+e^{-x}}{(1+e^{-x})^2}$$

Split the derivative in two terms:

$$\frac{d}{dx}S(x) = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$

Both terms have the S(x), so we can take that out and distribute it:

$$\frac{d}{dx}S(x) = \frac{1}{(1+e^{-x})}(1 - \frac{1}{1+e^{-x}})$$

Now we can realise, using the original value of the sigmoid function, that the expression can be represented as:

$$\frac{d}{dx}S(x) = S(x)(1 - S(x))$$

This is why in the problem we changed $\frac{da}{dz}$ to $a * (1 - a)$.

And finally, we go backwards to the expressions of w1, w2 and b, to find out how much we need to change the parameters of the logistic regression function:

$$\frac{dL(a,y)}{dw1} = \frac{dL(a,y)}{dz} * \frac{dz}{dw1} = (a - y) * (x1)$$ (="dw1")

$$\frac{dL(a,y)}{dw2} = \frac{dL(a,y)}{dz} * \frac{dz}{dw2} = (a - y) * (x2)$$ (="dw2")

$$\frac{dL(a,y)}{db} = \frac{dL(a,y)}{dz} * \frac{dz}{dwb} = (a - y) * (b)$$ (="db")

Gradient Descent on m Examples

In the previous topic, we saw how to compute derivatives and implement gradient descent with respect to just one training example for logistic regression. Now, we want to do it for m training examples. We need to deal with the cost function J:

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

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

As shown below, to compute the cost function's derivative value (considering just two features w1 and w2), we can apply the derivative to each term of the sum, and take the average between them.

$$\frac{d}{dw1}(J(w,\ b))\ =\ \frac{1}{m}\sum_{i=1}^{m} \frac{d}{dw1}L(a^{(i)},\ y^{(i)})$$

$$\frac{d}{dw2}(J(w,\ b))\ =\ \frac{1}{m}\sum_{i=1}^{m} \frac{d}{dw2}L(a^{(i)},\ y^{(i)})$$

These values will allow us to implement gradient descent. Let's make an algorithm to compute this value:

J=0; dw1=0; dw2=0; db=0
for i=1 to m

$$z^{(i)} = w^T x^{(i)} + b$$
$$a^{(i)} = \sigma(z^{(i)})$$
$$J \mathrel{+}= -(y^{(i)} log a^{(i)} + (1 - y^{(i)}) log(1 - a^{(i)}))$$
$$dz^{(i)} = a^{(i)} - y^{(i)}$$
$$dw1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$
$$dw2 \mathrel{+}= x_2^{(i)} dz^{(i)}$$
$$db \mathrel{+}= dz^{(i)}$$

J/=m ; dw1/=m ; dw2/=m ; db/=m //taking the average of the variables
w1 := w1 - α dw1
w2 := w2 - α dw2
b := b - α db // α is the learning rate
In this example, we've had just 2 features (w1 and w2). But if we need a high value of features, we might implement 2 for loops in the code - the first for loop runs over the m training examples, and the second for loop runs over the n features.

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 to n

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

J/=m ; dw1/=m ; dw2/=m ; db/=m //taking the average of the variables
w1 := w1 - α dw1
w2 := w2 - α dw2
b := b - α db // α is the learning rate

But the use of for loops can cause a bad performance to the algorithm (the complexity would be O(n^2)). To avoid this problem, we use a set of techniques called vectorization, that allow us to get rid of these explicit for-loops in the code. This technique is really helpful, due the fact that the datasets are becoming bigger and bigger every day.

Python and Vectorization

Vectorization

The vectorization is the art of getting rid of explicit for-loops in our code. In deep learning, we often find ourselves training neural networks using really large datasets. With vectorization, we can optimize our code a lot (in terms of time complexity).

In Python, a non vectorized code would be like this:
z=0
for i in range (n-x)
$z \mathrel{+}= w[i] * x[i]$
z += b

Using vectorization (in order to use this optimization method, we need to import the numpy library), we would have:
z = np.dot(w, x) + b

The term np.dot(w, x) is equivalent to $w^T x$.

An example written in python can be seen below:
In this example, we compare the running times of a code using vectorization (a technique capable of reducing the numbers of for-loops in the code) and the tradicional code, with for-loops. These code examples were taken from the Coursera Deep Learning course.

```
import numpy as np
import time

a = np.random.rand(1000000) #creates a random vector with 1M elements
b = np.random.rand(1000000) #creates a random vector with 1M elements

tic = time.time() #saves time to compute the running time of np.dot
c = np.dot(a,b)
toc = time.time() #saves time to compute the running time of np.dot

print(c)
print("Vectorized version:" + str(1000*(toc-tic)) + "ms")


c = 0
tic = time.time() #saves time to compute the running time of np.dot
for i in range(1000000):
    c += a[i]*b[i]
toc = time.time() #saves time to compute the running time of np.dot

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

Output:

```
250224.7398060972
Vectorized version:2.5985240936279297ms
250224.7398060912
For loop:788.5439395904541ms
```

As we can see, the vectorized version is way more optimized than the traditional one.

Vectorizing Logistic Regression

We can implement the vectorizing logistic regression using a single line of code. Let's assume we want to train a single neural network with m features (x1, x2, … xm). Each one of the features have n_X values. We'll need to compute the following equations:

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

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

and so on… (until the m features got computed)

Let's call Z a vector, that will contain all the z values. The vector X, as we already saw in the previous studies, is a matrix with (n_x, m) dimension (each column represents a different feature and n_x is the number of rows, that represents the input values). Here is a representation of Z:

$$Z = [z^{(1)}\ z^{(2)}\ ...\ z^{(m)}] = w^T X + [b\ b\ ...\ b] = [w^T x^{(1)} + b \quad w^T x^{(2)} + b \quad ... \quad w^T x^{(m)} + b]$$

To create the Z vector in Python, we only need a single line of code:

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

b is a real number, and with this code above it gets added in every element of the Z vector (this operation is called Broadcasting). Now we need to create a vector to represent the "a" values. Let's call this vector A:

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

We'll see more details of how to implement these calculations in the programming assignment.

Vectorizing Logistic Regression's Gradient Output

As we saw in the previous notes, one of the steps of the gradient descent is given by the following equations:

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

So let's create a vector, called dZ, to stack all dz elements and another two vectors A and Y, to stack a and y elements:

dZ = [$dz^{(1)} \quad dz^{(2)} \quad \dots \quad dz^{(m)}$]

$$A = [a^{(1)}\ a^{(2)}\ ...\ a^{(m)}]$$

$$Y = [y^{(1)}\ y^{(2)}\ ...\ y^{(m)}]$$

Analysing the vectors, we can conclude that:

$$dZ = A - Y$$

Now we need to vectorize the dw and db values. Let's start with dw:

$$dw = 0$$

$$dw\ += x^{(1)} dz^{(1)} \qquad dw\ += x^{(2)} dz^{(2)} \qquad \dots \qquad dw\ += x^{(m)} dz^{(m)}$$

And then:

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

Now let's compute db:

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

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

Let's put all together:

This code below is the original code, with for-loops:
J=0; dw1=0; dw2=0; db=0
for i=1 to m:

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

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

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

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

$$dw1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$

$$dw2 \mathrel{+}= x_2^{(i)} dz^{(i)}$$

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

J/=m , dw1/=m , dw2/=m , db/=m
w1 := w1 - α dw1
w2 := w2 - α dw2
b := b - α db // α is the learning rate

Now, the vectorized version:
Z = np.dot(w.T, X) + b
A = σ (Z)
dZ = A - Y

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

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

w := w - α dw
b := b - α db

Broadcasting in Python

Broadcasting is a Python technique that allows us to make our Python code to run faster. Let's create a matrix A, with 3 rows and 4 columns to exemplify the broadcasting use:

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

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

If we want to compute how much calories each food has, we can use the following command:
cal = A.sum(axis = 0)

This command allows us to sum all the elements of the column. If we want to compute the sum of the elements on the row, we change the code to:
var = A.sum(axis = 1)

Now if we want to calculate the % of calories for carbohydrates, protein and fat, without using for loops, we use the following command:
percentage = 100*A/cal.reshape(1,4)
This line of code allows us to divide the A matrix with a 1 by 4 matrix (the cal matrix, that represents the sum of calories of each food). The command “reshape” is an example of broadcasting, allowing us to divide matrices with different dimensions.

Another example of broadcasting:
Add one constant value to a vector:

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} + 100$$

In this case, what Python would do, is take the number 100 and expand it to a vector with the same size of the original vector. This is the result of the operation:

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

Another example:

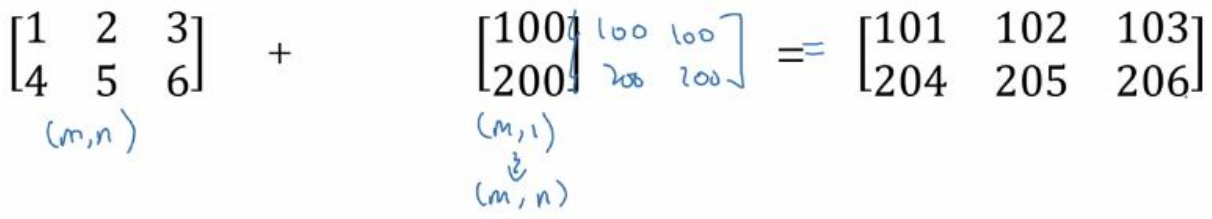
$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix} + [100 \quad 200 \quad 300]$$

In this case, Python would adapt the size of the second matrix, so they can be added:

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

And the last example:
If we want to add a 2x3 matrix with a 2x1 matrix:

The result would be:



General principle:
-If we have a (m,n) matrix, and + | - | / | * with a (1,n) matrix, the broadcasting will transform the second matrix to a (m,n) matrix (in this case, copying the value of the columns to all of the m rows).
-If we have a (m,n) matrix, and + | - | / | * with a (m,1) matrix, the broadcasting will transform the second matrix to a (m,n) matrix (in this case, copying the value of the rows to all of the n columns).
-If we have a (m,1) matrix, and + | - | / | * with a real number, the broadcasting will transform the number to a (m,1) matrix (in this case, copying the value of the real number to all the rows of the new matrix).

A note on Python/numpy vectors

Some tips and tricks to manipulate vectors in python:
a = np.random.randn(5)
a.shape = (5,)
This is called a “rank 1 array”, a data structure. This kind of array is not useful when implementing neural networks. Instead, use column vectors or row vectors:
a = np.random.randn(5,1) #row vector
a.shape = (5, 1)
a = np.random.randn(1, 5) #column vector
a.shape = (1, 5)

The use of column/row vectors avoid a lot of errors that could appear in the vectors manipulation.

To certify that the vector we are dealing with are in the expected dimension, we could use the following command:

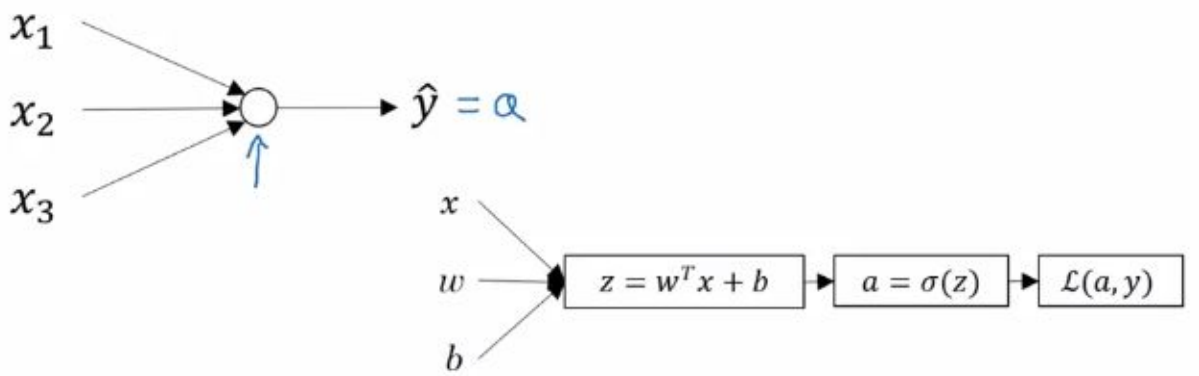
assert(a.shape == (5, 1)) #if “a” doesn’t have the size (5,1), an error - AssertionError is
                           #going to appear and the program is going to stop

And finally, if for some reason we end up with a rank 1 array, we can reshape it with the following command:
a = a.reshape((5,1))

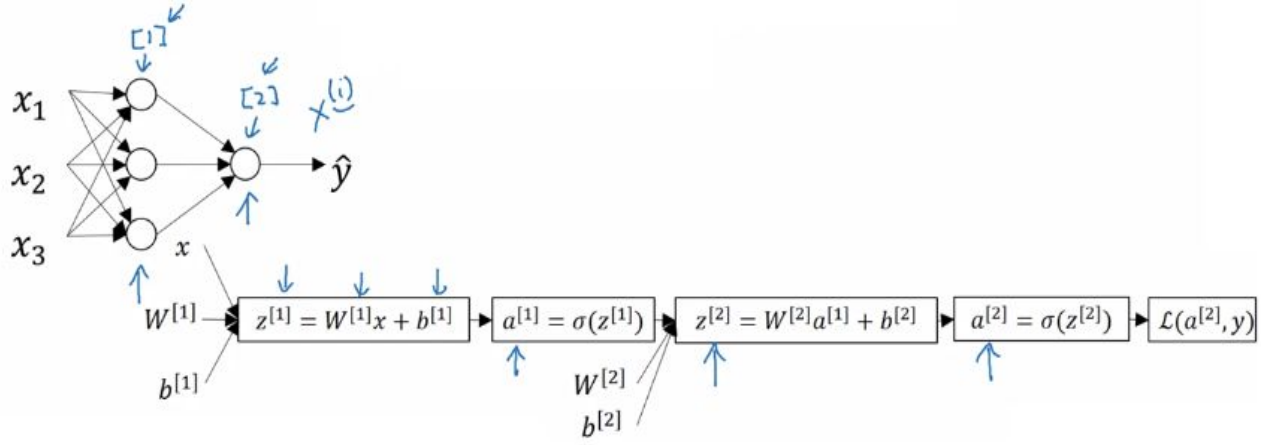
Shallow Neural Networks

Neural Networks Overview

As we saw in the previous week, a single neuron neural network is given by the following computation graph:



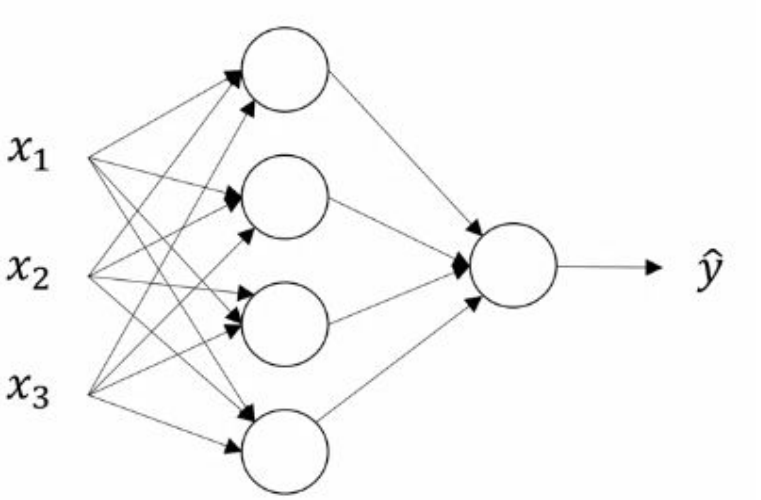
The circle represents the computation of z, a and the loss function values.
Let’s take a look inside a neural network with more than one neuron:



This neural network has 4 neurons. Each neuron computes a z and “a” value, and the final neuron computes the loss function value. The square brackets in the image [1] and [2] represents the deepness of the neuron in the neural network.

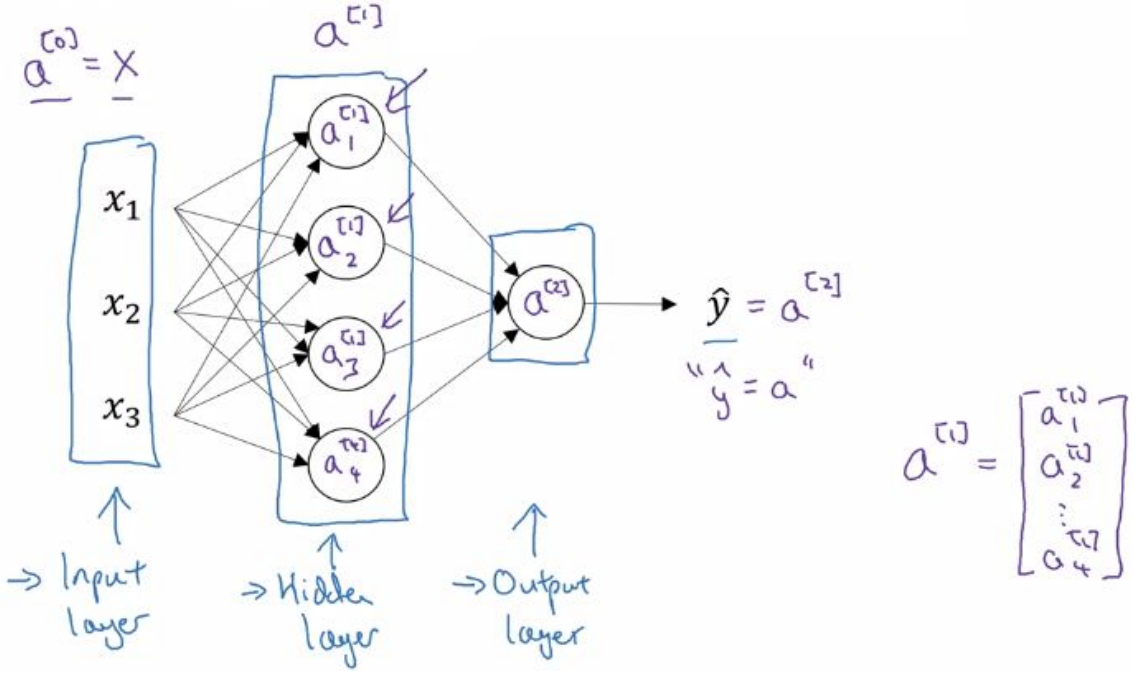
Neural Network Representation

To better understand the neural network representation, let's explain one representation:



In this image, we have the input features, x1, x2 and x3, stacked up vertically. This is called the input layer of the neural network. The first vertical stack of circles represents the hidden layer of the neural network. And finally, the last circle represents the output layer of the neural network, and it's responsible to generate the predict value $\hat{y}$.

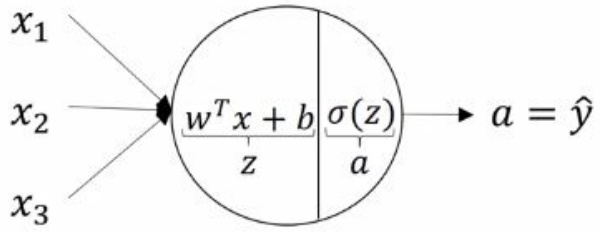
The notation "hidden layer" is used because in the training set there are only values of the inputs x and the target outputs y. So the term hidden layer refers to the fact that in a training set the true values for the nodes in the middle are not observed (you don't see what value they should be).



An alternative notation for the input features X vector is $a^{[0]}$. The term a corresponds to “activations”, and refers to the values that different layers of the neural network are passing on to the subsequent layers. The hidden layer then receives the input layer vector $a^{[0]}$, process these values and generate a $a^{[1]}$ vector, that is a column vector with all the values computed by the hidden layer neurons. The output layer then generates a $a^{[2]}$ vector, that is a real number (equivalent to $\hat{y} = a$ in the previous studies with a single neuron).
The example above is called a 2 layer neural network. The reason for this name is that we don’t count the input layer in the name.
Each layer has different w and b parameters. The hidden layer in this case will have $w^{[1]}$ and $b^{[1]}$ parameters, and the output layer will have $w^{[2]}$ and $b^{[2]}$ parameters (
Obs: in this case, the dimensions of each w and b parameters would be:
- $w^{[1]}$ is going to be a (4,3) dimension matrix (4 nodes and 3 input features);
- $b^{[1]}$ is going to be a (4,1) vector (4 constant bias values for each node);
- $w^{[2]}$ is going to be a (1,4) vector (1 value for the output layer and 4 values that come from the hidden layer);
- $b^{[2]}$ is going to be a (1,1) vector (1 bias value for the output node)).

Computing a Neural Network’s Output

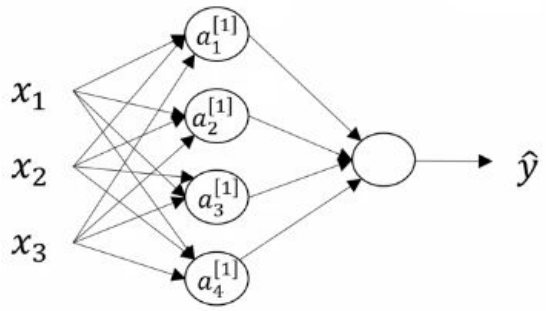
Each node of the neural network performs the following calculations:



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

$$a = \sigma(z)$$

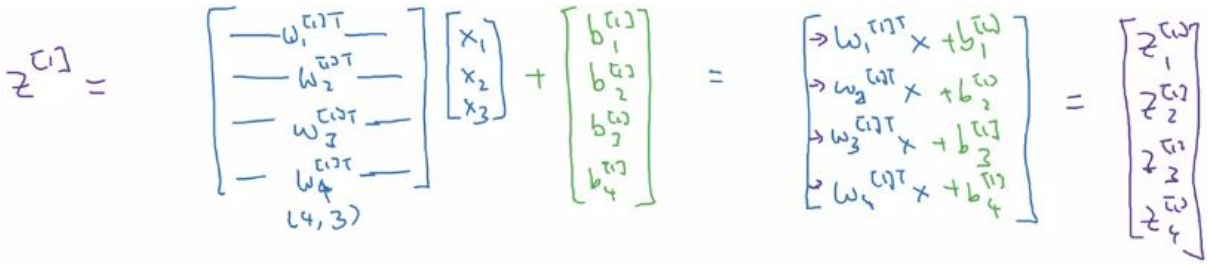
So the hidden layer nodes will compute the following equations:



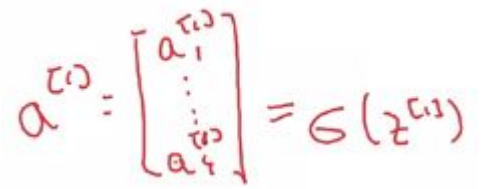
$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]},\ a_1^{[1]} = \sigma(z_1^{[1]})$$
$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]},\ a_2^{[1]} = \sigma(z_2^{[1]})$$
$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]},\ a_3^{[1]} = \sigma(z_3^{[1]})$$
$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]},\ a_4^{[1]} = \sigma(z_4^{[1]})$$

Our goal is to compute all the values of z:

As we can see, $w_n^{[1]T}$ is a vector. We can stack all the $w_n^{[1]T}$ vectors, form a matrix, multiply them by the input vector and add b. This is going to be our $z^{[1]}$ vector:



Adding another notation, the (4,3) matrix with the w values, will be called $W^{[1]}$ and the (4,1) vector with the b values will be called $b^{[1]}$. We have them computed the values of z in the hidden layer. The last thing we need to do is to compute the values of $a^{[1]}$ (the hidden layer activation vector):



Summarizing the informations above:

Given input x:

$$z^{[1]} = W^{[1]}x + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

If we want, we can call the x inputs $a^{[0]}$. In this mode, the equations would be like this:

Given input x:

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

$$a^{[1]} = \sigma(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[2]} = \sigma(z^{[2]})$$

The dimensions of the matrices and vectors would be like this:

$$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$$
$$(4,1) \quad (4,3) \quad (3,1) \quad (4,1)$$

$$a^{[1]} = \sigma(z^{[1]})$$
$$(4,1) \quad (4,1)$$

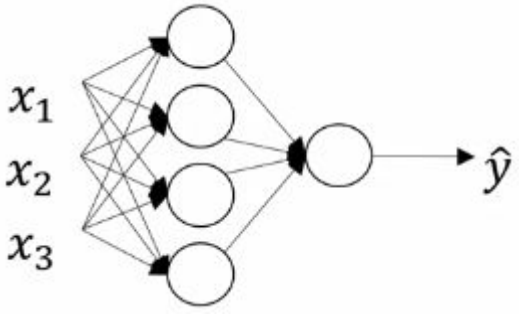
$$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$$
$$(1,1) \quad (1,4) \quad (4,1) \quad (1,1)$$

$$a^{[2]} = \sigma(z^{[2]})$$
$$(1,1) \quad (1,1)$$

To compute the neural network output, we only need these four equations.

Vectorizing across multiple examples

Let's analyze the neural network's output for a input vector X that has m training examples:



$X \text{ ----> } a^{[2]} = \hat{y}$:

$x^{(1)} \text{ ----> } a^{[2](1)} = \hat{y}^{(1)}$

$x^{(2)} \text{ ----> } a^{[2](2)} = \hat{y}^{(2)}$

$x^{(3)} \text{ ----> } a^{[2](3)} = \hat{y}^{(3)}$

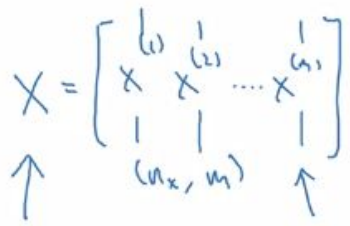
$x^{(4)} \text{ ----> } a^{[2](4)} = \hat{y}^{(4)}$

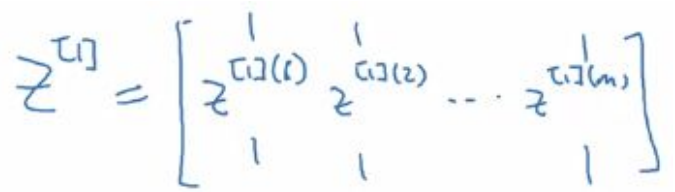
The notation for the activation values is going to be: $a^{[K](i)}$, where k represents the layer and i represents the example number. The code to calculate these values can be seen below:

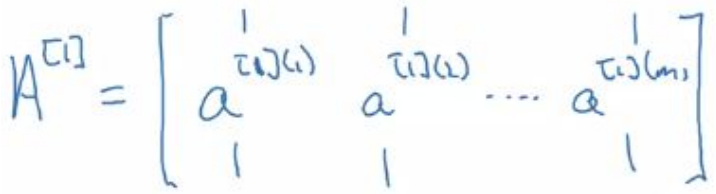
```
for i = 1 to m:
```

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

Now let's vectorize these pieces of code, to get rid of the for-loop. The steps we need to take, is to transform all the iterative values into single matrices. Using this strategy, we remove the necessity of using a for-loop. So:







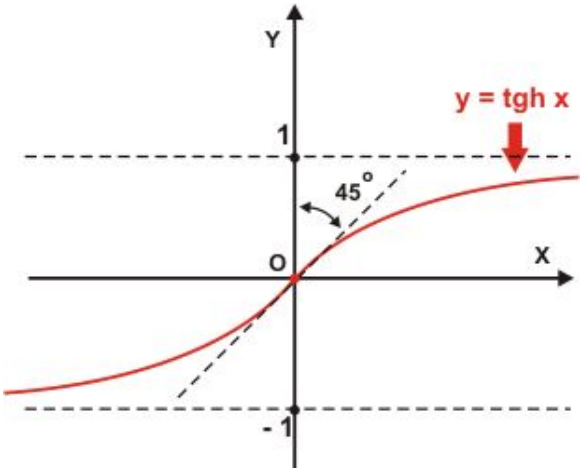
The same principle applies to the $Z^{[2]}$ and $A^{[2]}$ matrices. So, the vectorized version of the code is:

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

Activation Functions

When we build a neural network, one of the choices we've got to make is what activation function to use in the hidden layers. So far, we've just been using the

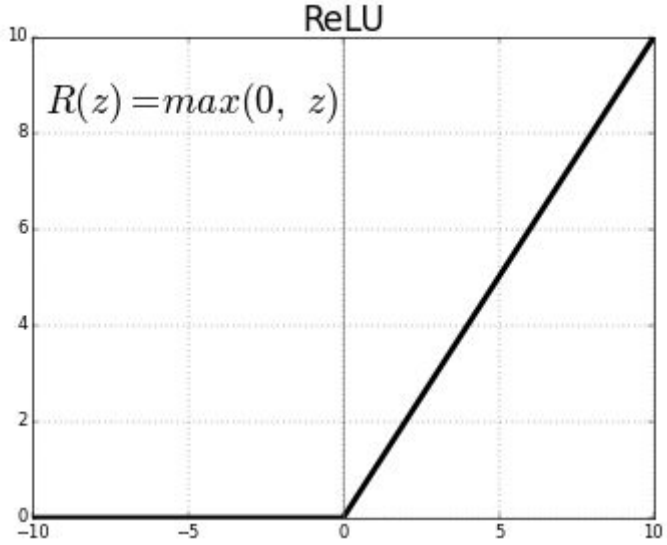
sigmoid activation function, but sometimes other choices can work much better. One of the functions is the tgh(x):



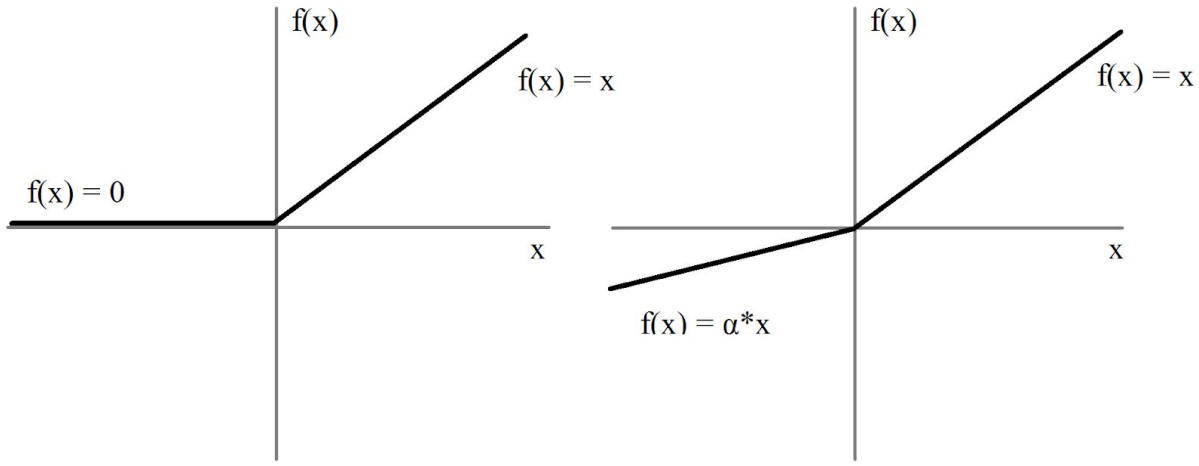
$$y = tghx \Rightarrow y = \frac{senhx}{\cosh x} \Rightarrow y = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

The tgh function is almost always superior compared to the sigmoid function. It happens because the tgh function centers the data (near 0) with a better performance compared to the sigmoid function (that centralizes the data near 0.5). However, the sigmoid function still has a better performance when it's used in the output layer, in cases we've got a binary classification problem (as $y \, \varepsilon \, \{0, 1\}$, it's helpful if the $\hat{y}$ is in the same range $\{0, 1\}$).

Another commonly used function is the ReLU (Rectified Linear Unit) function:
This function's derivative is 1, so long as z is positive, and is 0, when z is negative. When z=0, the derivative is not well defined, but in this case we can pretend it's either 1 or 0, and it'll work just fine.



One disadvantage of the ReLU function is: its derivative assumes a zero value when z is negative. This problem is solved using an adapted ReLU function, called Leaky ReLU:



(ReLU in the left and Leaky ReLU in the right)

The Leaky ReLU function has a different curve when z is negative, resulting in a non negative derivative value. This function usually works better than the normal ReLU, although it's not used as much in practice.

These days, people usually prefer the ReLU activation function because it works great in a big variety of cases. So if you don't know what activation function to use, just use the ReLU function. The ReLU and Leaky ReLU functions are very good because the value of the derivative is almost always bigger than zero, making the learning much faster.

So let's summarize the activation functions:
-Use the sigmoid function as the activation function of the output layer, if it is a binary classification problem;
-The tgh(z) activation function has a better performance in the hidden layer than the sigmoid function;
-If you don't know what activation function to use, just use the ReLU function and your application will probably work just fine.
(and of course, if you have time to test, test them all and see which one has the best performance)

Derivatives of activation functions

Let's start reminding the sigmoid function derivative:

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

$$\frac{d}{dx}S(x) = S(x)(1 - S(x))$$

Them, let's go to the tanh function derivative:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

It is now possible to derive using the rule of the quotient and the fact that:
derivative of e^x is e^x and
derivative of e^{-x} is $-e^{-x}$

So you have:

$$\frac{d}{dx}\tanh(x) = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2}$$

$$= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} = 1 - \tanh^2(x)$$

And finally, the ReLU function derivative:

The derivative is:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases}$$

And undefined in $x = 0$.

Ps: The derivative of the ReLU function is undefined at z=0, but we can assume that it is either 1 or 0 value and the application will work just fine.

(The Leaky ReLU function has the same derivative value of the ReLU function for positive z values. For negative z values, its derivative is going to be the slope of the curve -> 0.001, 0.01, α ...)

Gradient descent for Neural Networks

Let's analyse a neural network with the following parameters:
- $W^{[1]}$ (with $(n^{[1]}, n^{[0]})$ dimension -> $n^{[0]} = n_X$)
- $b^{[1]}$ (with $(n^{[1]}, 1)$ dimension)
- $W^{[2]}$ (with $(n^{[2]}, n^{[1]})$ dimension)
- $b^{[2]}$ (with $(n^{[2]}, 1)$ dimension)

where $n^{[0]} = n_X$ is the dimension of the input layer, $n^{[1]}$ is the dimension of the hidden layer, and $n^{[2]}$ is the dimension of the output layer.

The cost function for this parameters, considering a binary classification is given by:

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} * \sum_{i=1}^{m} L(\hat{y} - y)$$

And the gradient descent:

Repeat{

compute predicts $(\hat{y}^{(i)}, i = 1, ..., m)$

$dW^{[1]} = \frac{dJ}{dW^{[i]}}$, $db^{[1]} = \frac{dJ}{db^{[i]}}$, …

$W^{[1]} = W^{[1]} - \alpha dW^{[1]}$

$b^{[1]} = b^{[1]} - \alpha db^{[1]}$

$W^{[2]} = W^{[2]} - \alpha dW^{[2]}$

$b^{[2]} = b^{[2]} - \alpha db^{[2]}$

}

Formulas for Computing derivatives:

-Forward propagation:

$Z^{[1]} = W^{[1]}X + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = g^{[2]}(Z^{[2]})$

Where $g^{[1]}$ and $g^{[2]}$ are activation functions.

-Backpropagation:

$dZ^{[2]} = A^{[2]} - Y$

$dW^{[2]} = \frac{1}{m} dZ^{[2]} * A^{[1]T}$

$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$

$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$ (this is a element-wise matrix product, the dims of the matrix are equal)

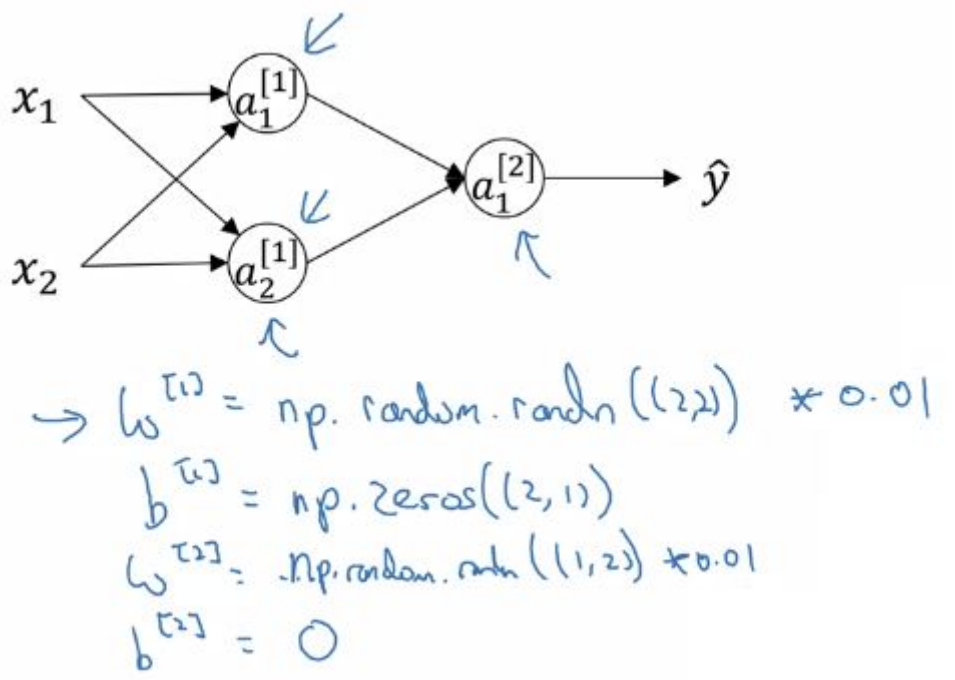
$dW^{[1]} = \frac{1}{m} dZ^{[1]} * X^{T}$

$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Random Initialization

When we change our neural network, it's important to initialize the weights randomly. For logistic regression, it was ok to initialize the weights to zero, but for a neural network initializing the parameters to zero and then applying gradient descent it won't work. The idea is: if we initialize the weights to zero, all the hidden units will be symmetric, and no matter how long we upgrade the center, all continue to compute exactly the same function. The parameter b doesn't have this problem, so we can

initialize it with zero values. The solution to this problem is simple, we just have to do a random initialization:



Note that the value of the weights are multiplied by 0.01. This small value usually prevents the activation functions to end up in the “flat” parts of the curve, accelerating the learning rate.

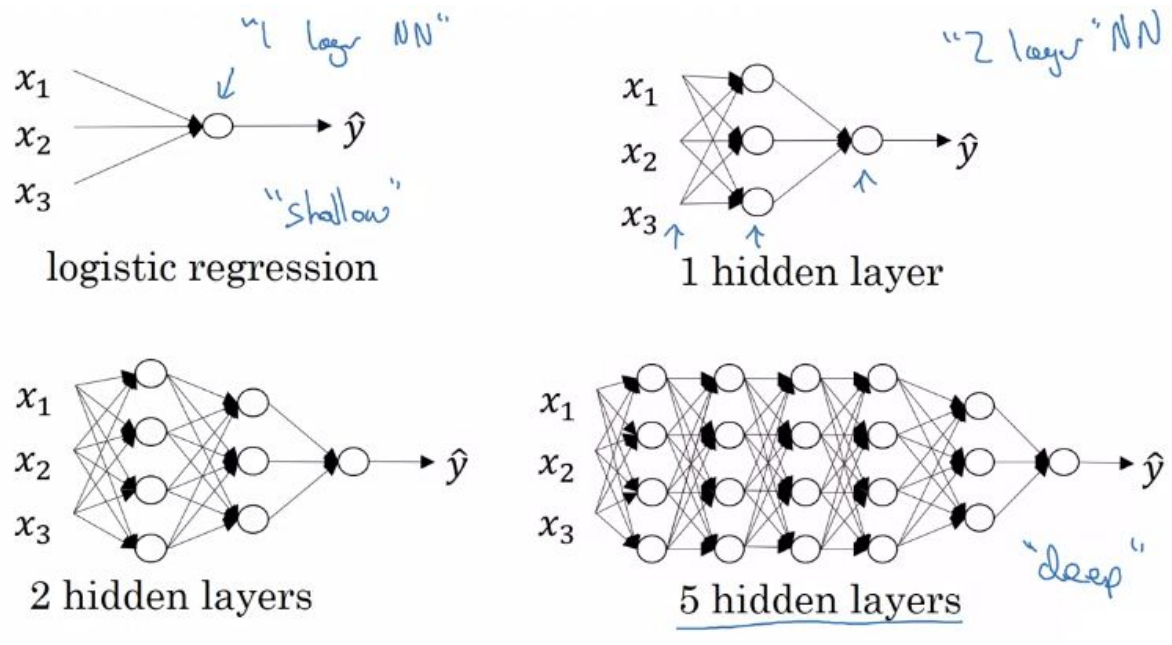
(Big values of z can slow down the learning process)

If you are training a very deep neural network, you may take a different constant than 0.01. We’ll see in the next week’s material when you might choose a different value.

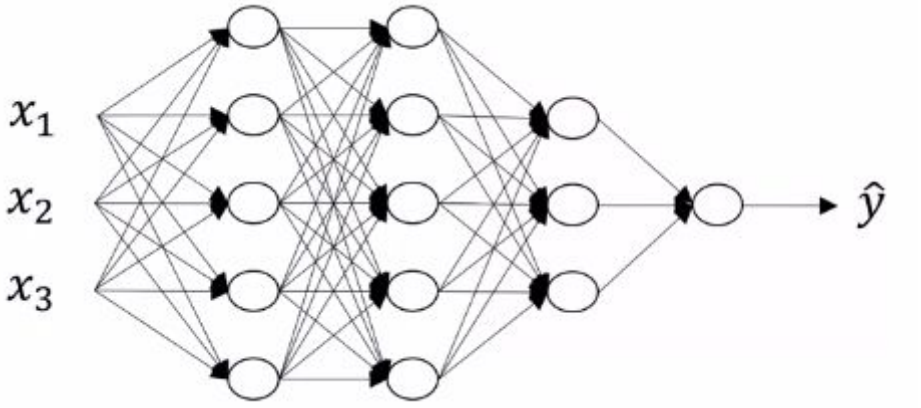
Deep Neural Network

Deep L-layer neural network

A neural network with one single layer is commonly called a “shallow model”. A deep neural network usually has 4 or more layers.



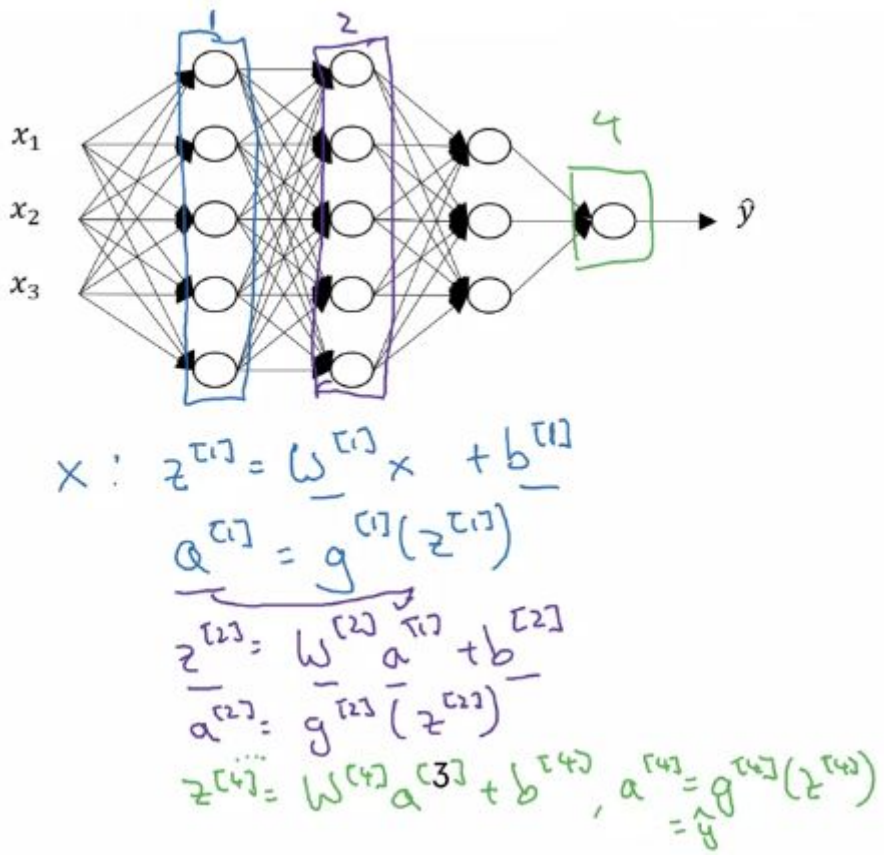
Let's take a look at some deep neural network notation:



In this case above, we use the notation L, to denote the number of layers (in this case L = 4). The notation $n^{[l]}$ denotes the number of units in each layer l (in this case $n^{[1]}$ = 5, $n^{[2]}$ = 5, $n^{[3]}$ = 3, $n^{[4]}$ =1 and $n^{[0]} = n_x = 3$). We also use the notation $a^{[l]}$ to denote the activations in layer l -> in forward propagation we use $a^{[l]} = g^{[l]}(z^{[l]})$. The notations $w^{[l]}$ and $b^{[l]}$ are used to denote the weights and biases for computing the value $z^{[l]}$ and the inputs and outputs are usually called $x = a^{[0]}$ and $\hat{y} = a^{[L]}$.

Forward Propagation in a Deep Network

In the forward propagation, we calculate z and a, for each layer of the neural network, using the parameters of the neural network. An example can be seen below:



The generalized formula to vectorized forward propagation is:

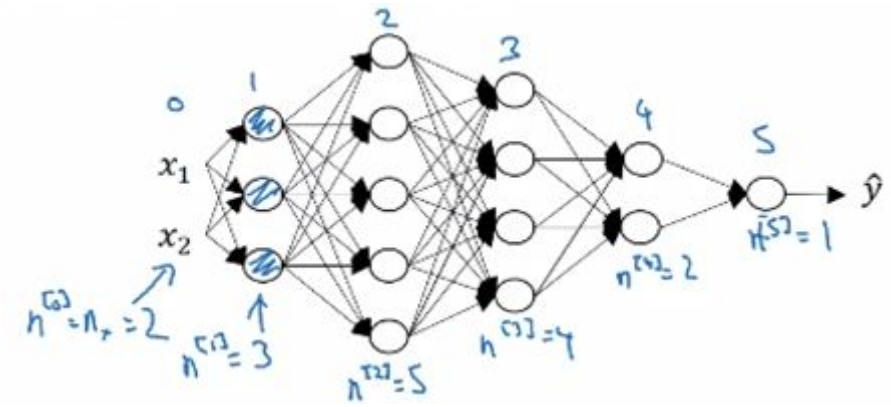
$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

In this case, it's ok to have a for-loop passing by all the layers of the neural network. The for loop will usually have a small number of iterations, and the optimization we've already done in the other steps are capable of keeping the code working fine.

Getting your matrix dimensions right

Checking the correctness of the matrices dimensions is a useful debugging tool. Let's see how to do that:



$$dim(z^{[l]}) = (n^{[l]}, 1)$$
$$dim(a^{[l]}) = (n^{[l]}, 1)$$
$$dim(w^{[l]}) = (n^{[l]}, n^{[l-1]})$$
$$dim(b^{[l]}) = (n^{[l]}, 1)$$

For example:
$$z^{[1]} = w^{[1]}.x + b^{[1]}$$
The dimensions:
$$dim(z^{[1]}) = (n^{[1]}, 1) = (3, 1)$$
$$dim(w^{[1]}) = (n^{[1]}, n^{[0]}) = (3, 2)$$
$$dim(x) = dim(a^{[0]}) = (n^{[0]}, 1) = (2, 1)$$
$$dim(b^{[1]}) = (n^{[1]}, 1) = (3, 1)$$

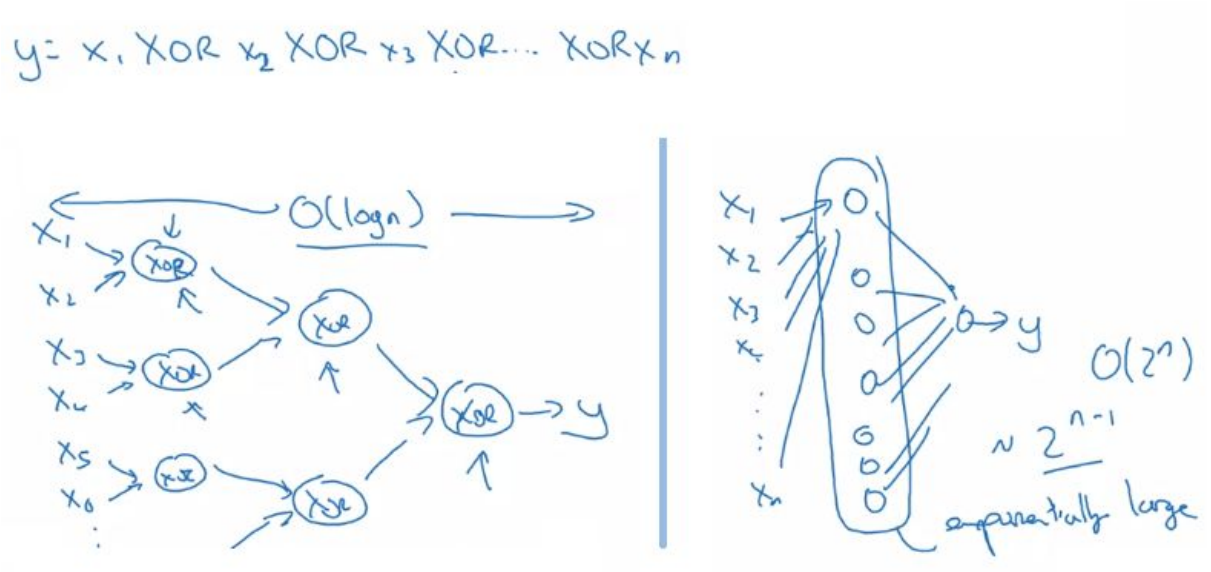
For backward propagation, the dimensions of dw and db are:
$$dim(dw^{[l]}) = dim(w^{[l]}) = (n^{[l]}, n^{[l-1]})$$
$$dim(db^{[l]}) = dim(b^{[l]}) = (n^{[l]}, 1)$$
$$dim(da^{[l]}) = dim(a^{[l]}) = (n^{[l]}, 1)$$
$$dim(dz^{[l]}) = dim(z^{[l]}) = (n^{[l]}, 1)$$

The matrices dimensions change when we use the vectorized implementation:
$$dim(Z^{[l]}) = (n^{[l]}, m)$$
$$dim(A^{[l]}) = (n^{[l]}, m)$$
$$dim(W^{[l]}) = (n^{[l]}, n^{[l-1]})$$
$$dim(b^{[l]}) = (n^{[l]}, 1) \rightarrow (n^{[l]}, m) \ (because\ of\ broadcasting)$$
$$dim(dZ^{[l]}) = dim(Z^{[l]}) = (n^{[l]}, m)$$
$$dim(dA^{[l]}) = dim(A^{[l]}) = (n^{[l]}, m)$$

The value of $dim(b^{[l]})$ remains the same, but when we add the bias in the Z equation, it gets added to all the elements of the matrix because of broadcasting.

Why deep representations?
Deep neural networks can compute lots of mathematical functions much easier compared to shallow neural networks. This is called the Circuit theory, informally, this theory says:
“There are functions you can compute with a “small” L-layer deep neural network that shallower networks require exponentially more hidden units to compute.”

An example we can see, is in the calculation of the XOR value of n inputs x:



In this example, if we use a deeper neural network, the complexity would be O(logn), otherwise, if we use a shallow neural network, the complexity would be exponential. So if you have the opportunity to use deeper neural networks it might result in a better performance.

Building blocks of deep neural networks

Starting in a random layer l, let's build blocks to represent the forward and backward propagation functions:

Layer l parameters: $w^{[l]}$, $b^{[l]}$
Forward function (without vectorization):
Input -> $a^{[l-1]}$ | Output -> $a^{[l]}$

computes $z^{[l]} = w^{[l]}.a^{[l-1]} + b^{[l]}$
computes $a^{[l]} = g^{[l]}(z^{[l]})$
store in cache $z^{[l]}$

Forward function (with vectorization):
Input -> $A^{[l-1]}$ | Output -> $A^{[l]}$

computes $Z^{[l]} = W^{[l]}.A^{[l-1]} + b^{[l]}$
computes $A^{[l]} = g^{[l]}(Z^{[l]})$
store in cache $Z^{[l]}$

Backward function (without vectorization):

Input -> $da^{[l]}$ | Output -> $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

uses cached $z^{[l]}$

computes $dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$

computes $dw^{[l]} = dz^{[l]}.\ a^{[l-1]^T}$

computes $db^{[l]} = dz^{[l]}$

computes $da^{[l-1]} = w^{[l]^T}.\ dz^{[l]}$

Obs: to make a function of dz, depending on the previous dz, we take the equation (4) and put it in the equation (1):

computes $dz^{[l]} = w^{[l-1]^T}.\ dz^{[l-1]} * g^{[l]'}(z^{[l]})$

Backward function (with vectorization):

Input -> $dA^{[l]}$ | Output -> $dA^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

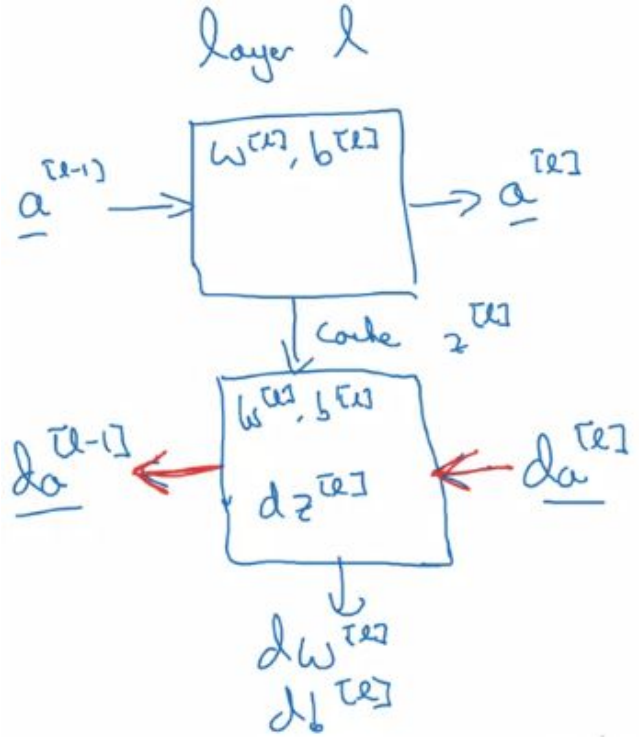
uses cached $z^{[l]}$

computes $dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$

computes $dW^{[l]} = \frac{1}{m}\ dZ^{[l]}.\ A^{[l-1]^T}$

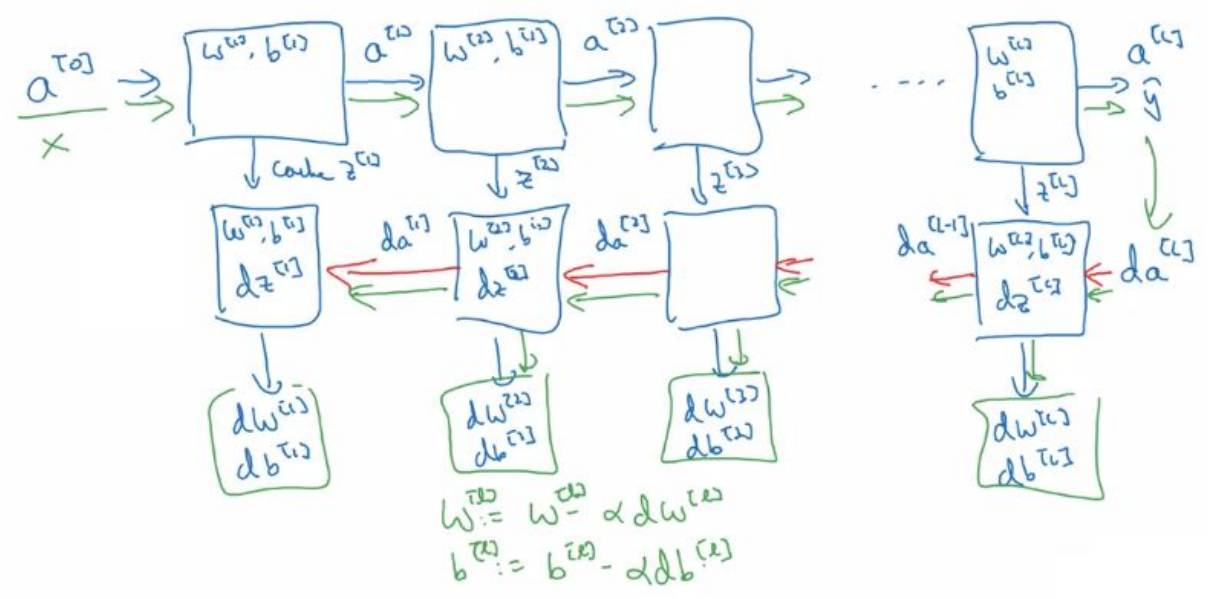
computes $db^{[l]} = \frac{1}{m}\ np.sum(dZ^{[l]},\ axis = 1,\ keepdims\ =\ True)$

computes $dA^{[l-1]} = W^{[l]^T}.\ dZ^{[l]}$



In this image, the first block corresponds to the forward block, and the second block corresponds to the backwards block. You can see that the inputs and outputs are

equivalent to those seen in the pseudocode above the image. There is an image below representing the blocks for a multilayer neural network:



Parameters vs. Hyperparameters

Being effective in your deep neural network requires you to organize not only your parameters, but also your hyperparameters. Let's see what are the hyperparameters:

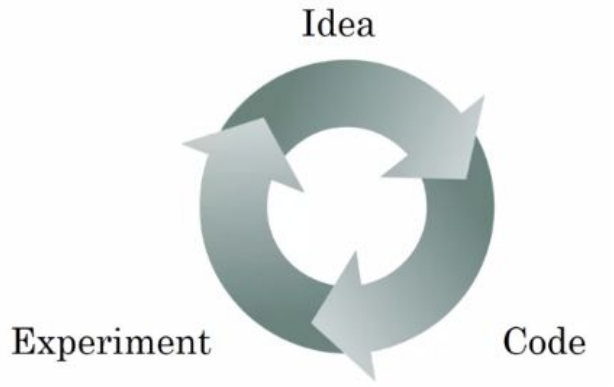
Parameters:
- W (weight)
- b (bias)

Hyperparameters:
- Learning rate α
- # iterations of the gradient descent,
- # hidden layers L
- # hidden units ($n^{[1]}$, $n^{[2]}$, ...)
- Choice of activation functions
- …
- Later on this course, we'll see new hyperparameters like: Momentum, mini-batch size, regularization parameters...

The hyperparameters are parameters that control the parameters W and b.

The choice of the hyperparameters is still a very empyric process. You might choose a certain value for the α parameter, as an example, and then discover better values

through experimentation. This is a cyclical process and requires a good time to make a good deep learning model.



Summary

Forward Propagation equations:

$$
\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]}) \\
&\vdots \\
A^{[L]} &= g^{[L]}(Z^{[L]}) = \hat{Y}
\end{aligned}
$$

Backward Propagation equations:

$$dZ^{[L]} = A^{[L]} - Y$$

$$dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L-1]^T}$$

$$db^{[L]} = \frac{1}{m} np.sum(dZ^{[L]}, axis = 1, keepdims = True)$$

$$dZ^{[L-1]} = W^{[L]^T} dZ^{[L]} * g'^{[L-1]}(Z^{[L-1]})$$

Note that * denotes element-wise multiplication)

$$\vdots$$

$$dZ^{[1]} = W^{[2]} dZ^{[2]} * g'^{[1]}(Z^{[1]})$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]^T}$$

Note that $A^{[0]^T}$ is another way to denote the input features, which is also written as X^T

$$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$$