

Deep Learning Specialization

by DeepLearning.AI

Course #1: Neural Networks and Deep Learning



[Course Site](#)

Made By: [Matias Borghi](#)

Table of Contents

Summary	3
Week 1 Introduction to deep learning	5
Introduction to Deep Learning	5
¿Qué es una red neuronal?	5
Aprendizaje Supervisado con Redes Neuronales	6
¿Por qué está despegando Deep Learning?	8
Week 2 Neural Networks Basics	9
Logistic Regression as a Neural Network	9
Fundamentos de la programación de redes neuronales	9
Clasificación binaria	9
Notación	9
Regresión logística	10
Casos extremos	11
Función de coste de la regresión logística	11
Gradiente descendente	12
¿Cómo se minimiza $J(w,b)$?	12
Gráfico computacional	13
Derivadas en un gráfico computacional	14
Gradiente descendente en regresión logística	14
Gradiente descendente en m ejemplos	15
Python and Vectorization	16
Vectorización	16
Implementación vectorial del gradiente descendente en regresión logística	16
Broadcasting en Python	17
Python/ Numpy vectors	17
Week 3: Shallow Neural Network	19
Representación de Redes Neuronales	19
¿Cómo se computan las salidas?	19
Introduciendo un poco de notación	21
Explicación de la implementación vectorizada	22
Funciones de activación	23
Ventajas y desventajas de las funciones de activación	24
Derivadas de las funciones de activación	24
Gradiente descendente para redes neuronales	25
Intuición de la backward propagation	27
Inicialización parámetros	28
Inicialización aleatoria	28
Week 4 Deep Neural Network	30
Redes neuronales profundas de L-capas	30

Forward and Backward propagation in deep neural networks	31
Resumen	32
¿Por qué las redes neuronales profundas generalmente funcionan mejor que las que no lo son?	33
¿Qué son los hiper parámetros?	34
Forward y Backward propagation	35

Summary

Week 1 Introduction to deep learning

Be able to explain the major trends driving the rise of deep learning, and understand where and how it is applied today.

Learning Objectives

- Discuss the major trends driving the rise of deep learning.
- Explain how deep learning is applied to supervised learning
- List the major categories of models (CNNs, RNNs, etc.), and when they should be applied
- Assess appropriate use cases for deep learning

Week 2 Neural Networks Basics

Learn to set up a machine learning problem with a neural network mindset. Learn to use vectorization to speed up your models.

Learning Objectives

- Build a logistic regression model structured as a shallow neural network
- Build the general architecture of a learning algorithm, including parameter initialization, cost function and gradient calculation, and optimization implementation (gradient descent)
- Implement computationally efficient and highly vectorized versions of models
- Compute derivatives for logistic regression, using a backpropagation mindset
- Use Numpy functions and Numpy matrix/vector operations
- Work with iPython Notebooks
- Implement vectorization across multiple training examples

Week 3 Shallow neural networks

Learn to build a neural network with one hidden layer, using forward propagation and backpropagation.

Learning Objectives

- Describe hidden units and hidden layers
- Use units with a non-linear activation function, such as tanh
- Implement forward and backward propagation
- Apply random initialization to your neural network
- Increase fluency in Deep Learning notations and Neural Network Representations
- Implement a 2-class classification neural network with a single hidden layer

Week 4 Deep Neural Networks

Understand the key computations underlying deep learning, use them to build and train deep neural networks, and apply it to computer vision.

Learning Objectives

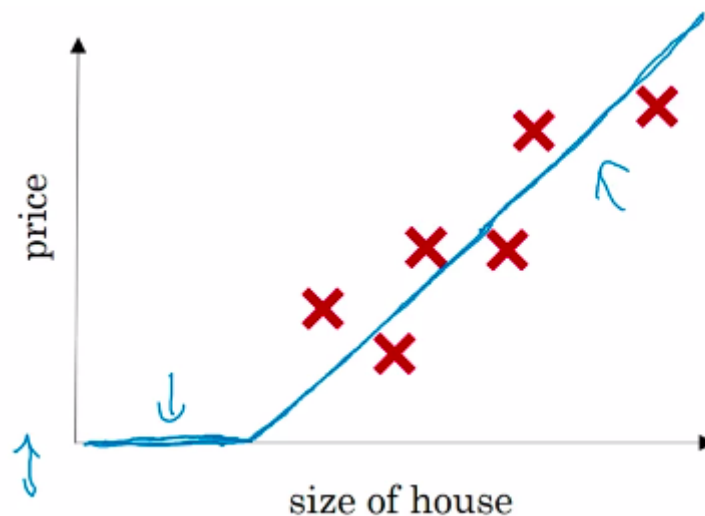
- Describe the successive block structure of a deep neural network
- Build a deep L-layer neural network
- Analyze matrix and vector dimensions to check neural network implementations
- Use a cache to pass information from forward to back propagation
- Explain the role of hyperparameters in deep learning

Week 1 Introduction to deep learning

Introduction to Deep Learning

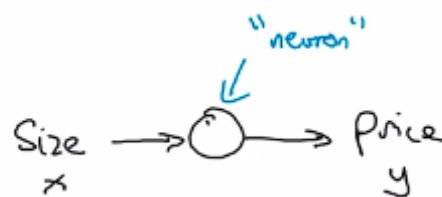
¿Qué es una red neuronal?

Predicción aplicado al precio de las casas.



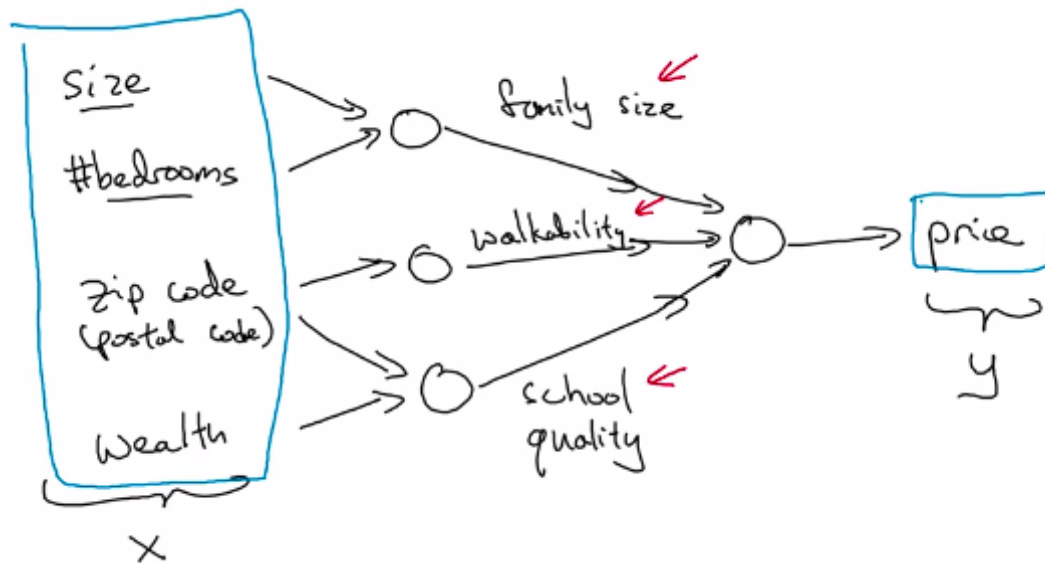
Los datos parecen ajustarse con un ajuste lineal. Dado que los precios no pueden ser negativos podemos agregar una sección donde el ajuste sería cero.

A esta función que predice los precios se la puede pensar como una red neuronal simple.



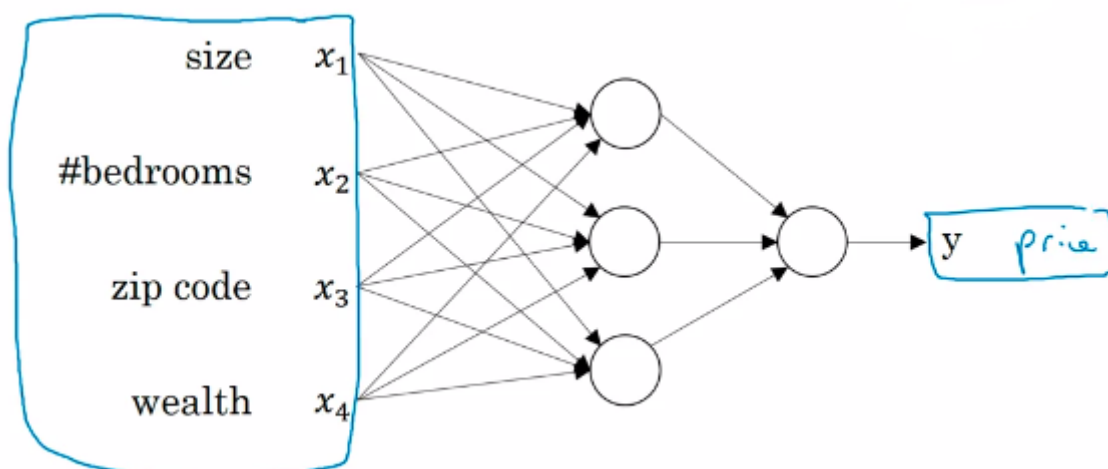
Este tipo de funciones que es cero y luego lineal es llamada RELU (Rectified Linear Unit). Una red neuronal más compleja se podría armar uniendo "neuronas" simples como la recién creada.

En lugar de crear una red neuronal que predice el precio de una casa en base al tamaño, también lo hace en función de otras características (features) como el número de habitaciones. También el tamaño de la familia (4 integrantes, 5 integrantes etc.).



Lo bueno es que, en un modelo de redes neuronales, no necesitamos conocer las características intermedias, sino que basta con determinar x e y .

En la realidad la red neuronal se implementa de la siguiente manera:



donde las unidades ocultas se conectan con todas las unidades precedentes.

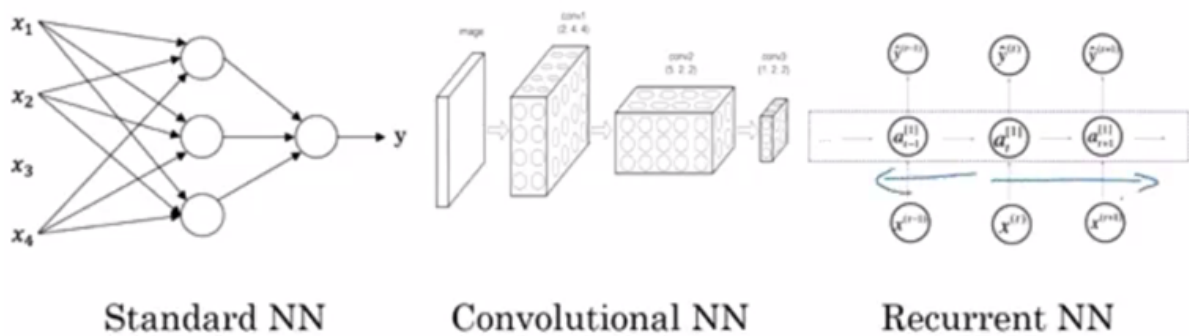
Aprendizaje Supervisado con Redes Neuronales

Ejemplos de Aprendizaje supervisado se muestran en la tabla siguiente:

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

donde cada aplicación en particular requiere una red neuronal específica ya sea CNN, RNN, híbridas, etc.

Un diagrama de estas redes neuronales podría ser el siguiente




Aplicaciones de Machine Learning sobre datos estructurados o no estructurados

Structured Data


Size	#bedrooms	...	Price (1000\$)
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



Audio

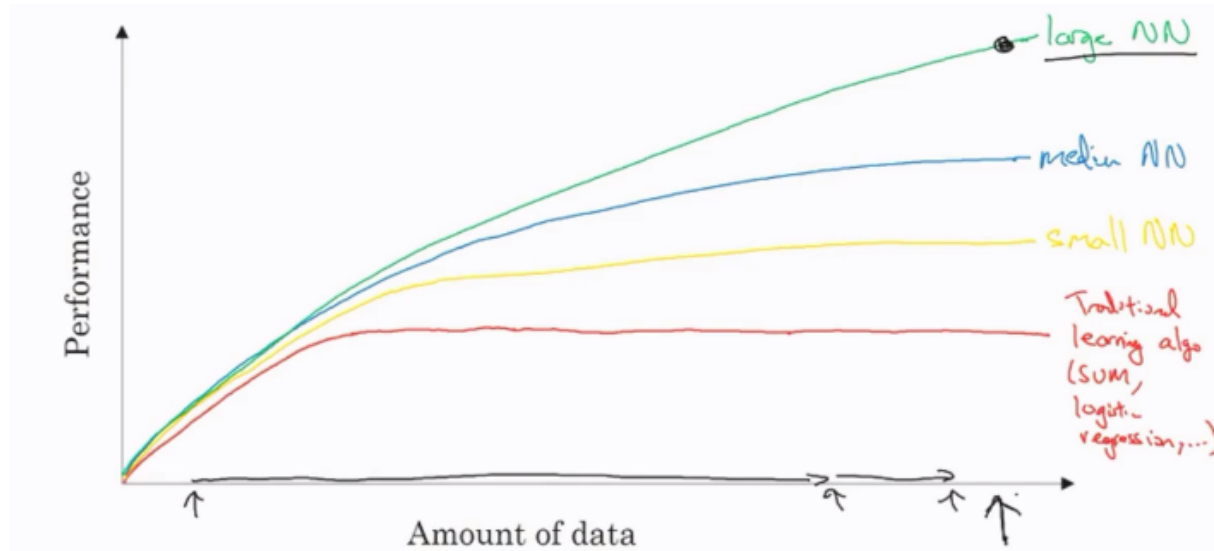


Image

Four scores and seven years ago...

Text

¿Por qué está despegando Deep Learning?



La cantidad de información disponible ha aumentado considerablemente. Los algoritmos tradicionales de aprendizaje no mejoran su rendimiento dada la mejora de datos, cosa que las redes neuronales si lo hacen.

Para obtener un rendimiento alto habría que tener una red neuronal grande, con muchas unidades ocultas y mucha información.

Week 2 Neural Networks Basics

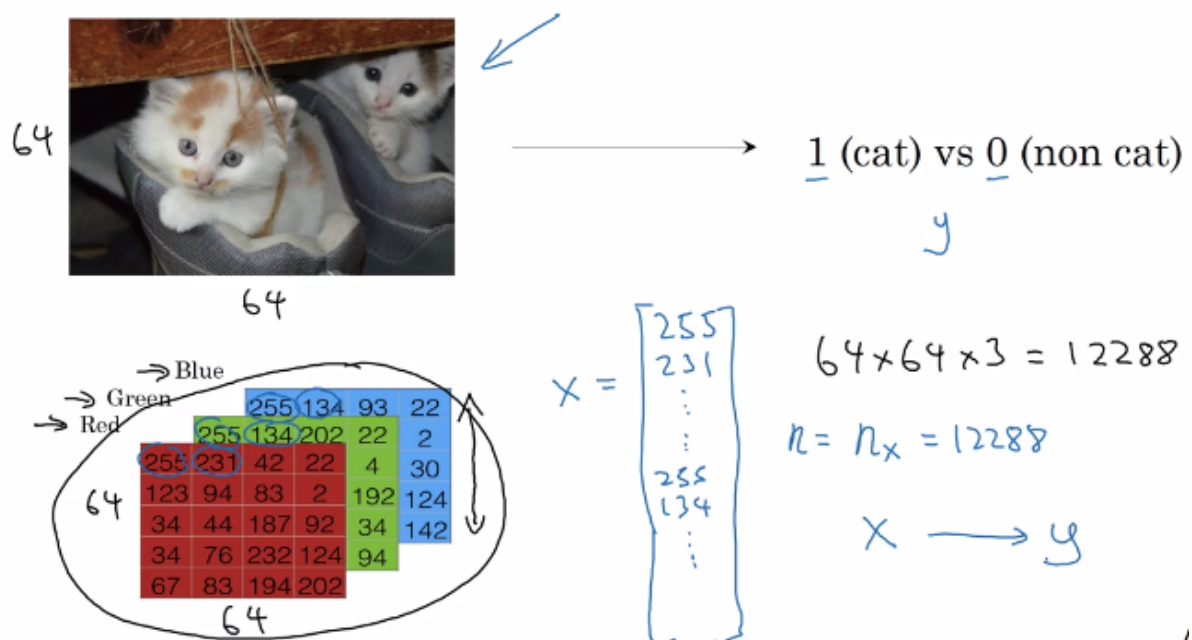
Logistic Regression as a Neural Network

Fundamentos de la programación de redes neuronales

Clasificación binaria

Un ejemplo sería determinar si, dada una foto, la misma pertenece a un gato o no.

Una foto color se puede representar como tres matrices RGB de, por ejemplo, 64x64. El vector de entrada (input feature) X tendrá como longitud $64 \times 64 \times 3 = 12288$. La idea del algoritmo de clasificación binaria es dado este vector X determinar si es gato (1) o no (0).



Notación

Ejemplo de entrenamiento representado por el par (x, y) . En total se tendrán m , conocido como el conjunto de ejemplos de entrenamiento (training set). La matriz X consiste en poner

en columnas los distintos ejemplos de entrenamiento. Con las salidas Y se procede de la misma manera.

$$(x, y) \quad x \in \mathbb{R}^{n_x}, y \in \{0, 1\}$$

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

$$M = M_{\text{train}} \quad M_{\text{test}} = \# \text{test examples.}$$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \begin{matrix} \uparrow \\ n_x \\ \downarrow \end{matrix}$$

$$X \in \mathbb{R}^{n_x \times m} \quad X.\text{shape} = (n_x, m)$$

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

$$Y \in \mathbb{R}^{1 \times m}$$

$$Y.\text{shape} = (1, m)$$

Regresión logística

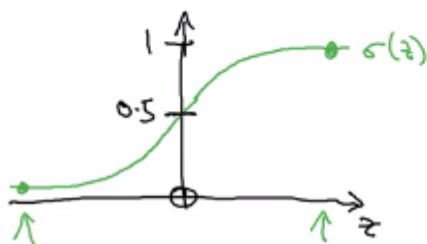
La idea es determinar w y b . Como z puede tomar cualquier valor real, se le aplica la función sigmoide para que tome valores entre 0 y 1, correspondientes a una probabilidad.

$$\text{Given } x, \text{ want } \hat{y} = \frac{P(y=1|x)}{0 \leq \hat{y} \leq 1}$$

$$x \in \mathbb{R}^{n_x}$$

$$\text{Parameters: } \boxed{w} \in \mathbb{R}^{n_x}, \boxed{b} \in \mathbb{R}.$$

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



Casos extremos

La notación del curso de Machine Learning de Andrew Ng no se utilizará.

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

$$\text{If } z \text{ large } \sigma(z) \approx \frac{1}{1+0} = 1$$

If z large negative number

$$\sigma(z) = \frac{1}{1+e^{-z}} \approx \frac{1}{1+\text{Big num}} \approx 0$$

$$x_0 = 1, \quad x \in \mathbb{R}^{n_x+1}$$

$$\hat{y} = \sigma(\theta^T x)$$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_{n_x} \end{bmatrix} \begin{matrix} \} b \leftarrow \\ \\ \\ \\ \} w \leftarrow \end{matrix}$$

Función de coste de la regresión logística

Para entrenar los parámetros w y b se necesita una función de coste.

La función de pérdida (Loss function) está asociada a un ejemplo únicamente. Puede definirse como el error cuadrado, pero no es convexa. Por eso se define de otra manera.

La función de coste se aplica sobre todo al conjunto de entrenamiento, minimizando la función para cada uno de estos puntos. Los valores w y b que minimizan la función serán los de interés.

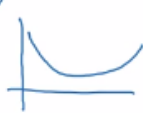
A continuación se verá aplicada a una regresión logística como una red neuronal muy pequeña.

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

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$, want $\hat{y}^{(i)} \approx y^{(i)}$. $\begin{matrix} x^{(i)} \\ y^{(i)} \\ z^{(i)} \end{matrix}$ i -th example.

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

$\mathcal{L}(\hat{y}, y) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})] \leftarrow$



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

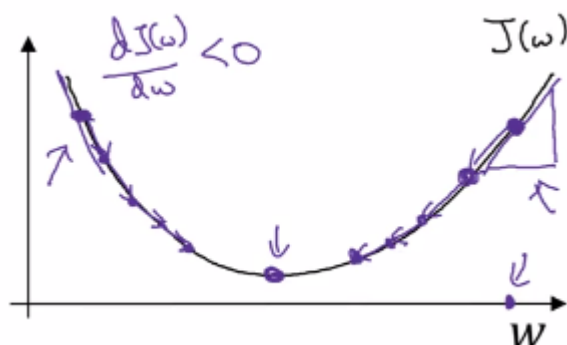
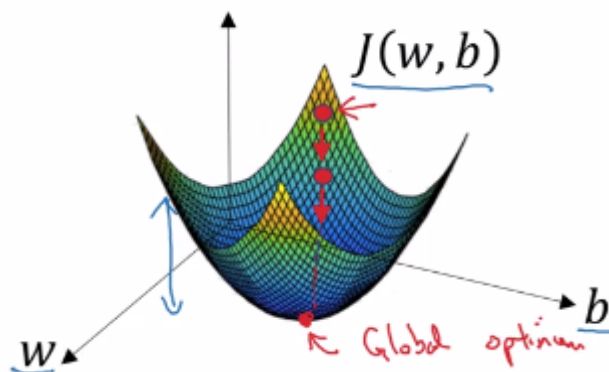
If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y}) \leftarrow$ Want $\log(1-\hat{y})$ large ... want \hat{y} small

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

Gradiente descendente

¿Cómo se minimiza $J(w, b)$?

Pensando el problema para un b fijo,



el algoritmo consiste en repetir la ecuación hasta que w o b no varíen drásticamente.

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

learning rate

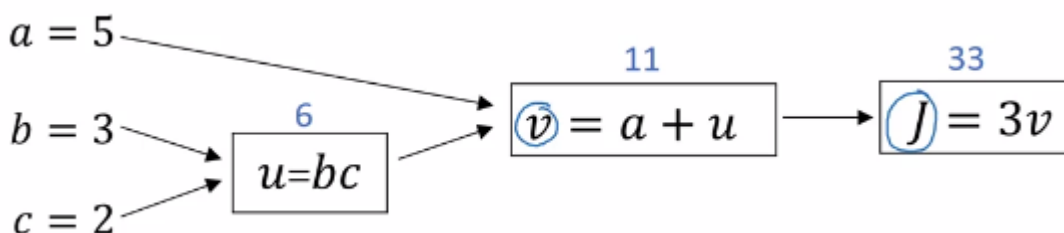
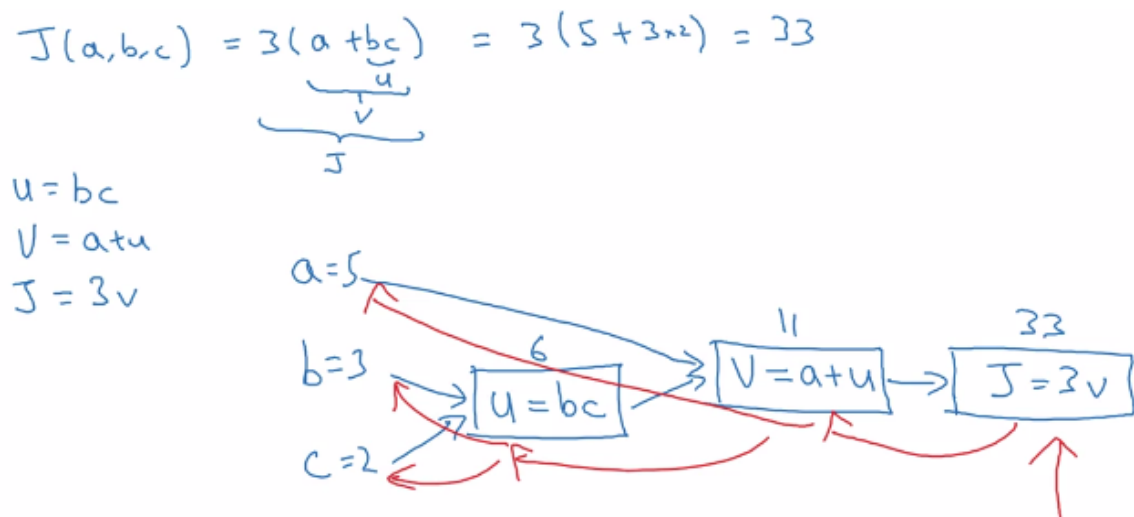
$$w := w - \alpha \frac{dJ(w,b)}{dw}$$

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

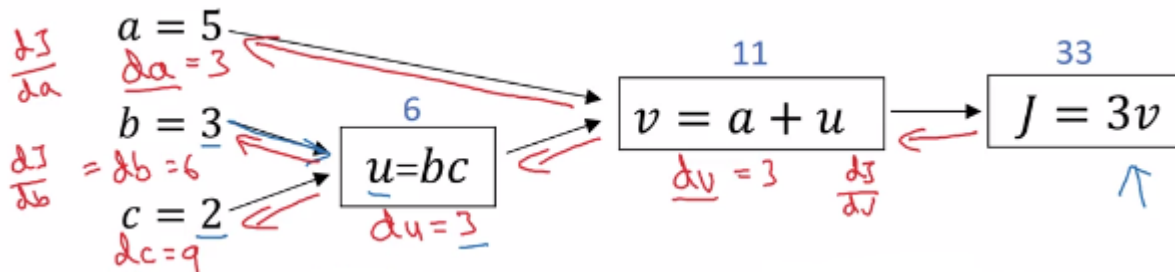
Gráfico computacional

Simplemente para tener una idea de back y forward propagation, veamos con un ejemplo como una función de varias variables puede ser evaluada hacia adelante “forward” mientras que para derivarla habría que hacerlo hacia atrás (“backward”).

En un ejemplo la función J depende de tres variables a , b y c . Las flechas azules representan la propagación hacia adelante para evaluar la función (cálculo de u y v), mientras que las rojas representan las derivadas (gradiente descendente).



Derivadas en un gráfico computacional



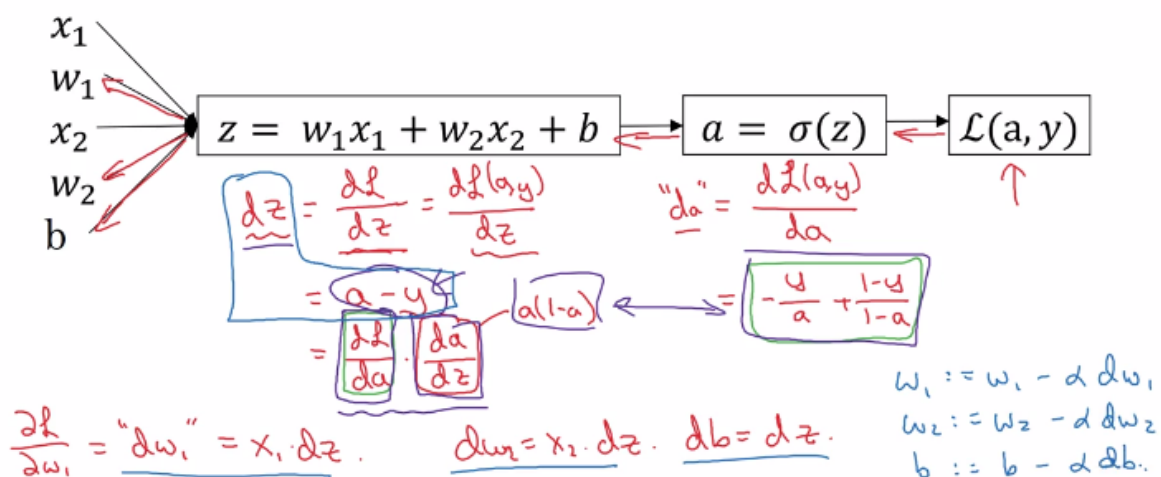
Gradiente descendente en regresión logística

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

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

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

Considerando dos entradas x_1, x_2 , para calcular w_1, w_2 y b primero habría que calcular dz y luego multiplicar por su correspondiente entrada.



Gradiente descendente en m ejemplos

En lugar de hacer lo anterior sobre la función de pérdida, debemos hacerlo sobre la función de coste.

Pero las derivadas toman una forma simple como el promedio sobre todos los ejemplos.

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

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

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

El algoritmo se puede escribir de la siguiente manera

$J = 0; \underline{dw_1} = 0; \underline{dw_2} = 0; \underline{db} = 0$ <u>For</u> $i = 1$ <u>to</u> m $z^{(i)} = w^T x^{(i)} + b$ $a^{(i)} = \sigma(z^{(i)})$ $J_t = -[y^{(i)} \log a^{(i)} + (1 - y^{(i)}) \log(1 - a^{(i)})]$ $\underline{dz^{(i)}} = a^{(i)} - y^{(i)}$ $\begin{matrix} \uparrow \\ dw_1 \end{matrix} + = x_1^{(i)} dz^{(i)}$ $\begin{matrix} \uparrow \\ dw_2 \end{matrix} + = x_2^{(i)} dz^{(i)}$ $\begin{matrix} \uparrow \\ db \end{matrix} + = dz^{(i)}$ $J /= m \leftarrow$ $\begin{matrix} \uparrow \\ dw_1 \end{matrix} /= m; \begin{matrix} \uparrow \\ dw_2 \end{matrix} /= m; \begin{matrix} \uparrow \\ db \end{matrix} /= m. \leftarrow$	$dw_1 = \frac{\partial J}{\partial w_1}$ $w_1 := w_1 - \alpha \underline{dw_1}$ $w_2 := w_2 - \alpha \underline{dw_2}$ $b := b - \alpha \underline{db}$
---	--

representando un único paso. Esto se repetirá hasta que tanto w como b converjan a un valor constante.

En este algoritmo tenemos dos for loops. Para que sea más eficiente se debería vectorizar.

Python and Vectorization

Vectorización

$$w = \begin{bmatrix} : \\ : \\ : \end{bmatrix} \quad x = \begin{bmatrix} : \\ : \\ : \end{bmatrix} \quad \begin{matrix} w \in \mathbb{R}^{n_x} \\ x \in \mathbb{R}^{n_x} \end{matrix}$$

Non-vectorized:

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

Vectorized

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

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

Un ejemplo hecho por Andrew Ng demuestra la diferencia entre vectorizar o no el algoritmo.

Implementación vectorial del gradiente descendente en regresión logística

Todavía se necesitaría un for-loop sobre el número de iteraciones que se quieren para minimizar la función de coste.

$$\begin{aligned}
z &= w^T X + b \\
&= \text{np.dot}(w.T, X) + b \\
A &= \sigma(z) \\
dz &= A - Y \\
dw &= \frac{1}{m} X dz^T \\
db &= \frac{1}{m} \text{np.sum}(dz) \\
w &:= w - \alpha dw \\
b &:= b - \alpha db
\end{aligned}$$

Broadcasting en Python

Es una técnica de Python para hacer funcionar el código más rápido.

A modo de ejemplo se quiere pasar la matriz a una matriz de porcentajes. Para ello primero conviene sumar sobre cada columna los valores y dividir ese valor total sobre cada elemento de la matriz (y multiplicar por 100).

	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

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

donde axis=0 indica sumar las por columna.

En porcentaje se dividió A que tiene dimensión 3x4 por cal que tiene dimensión 1x4.

Python/ Numpy vectors

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

} Don't use

No son recomendables los arrays de rank 1 porque, por ejemplo, al usar `np.dot()` el comportamiento no es el esperado.

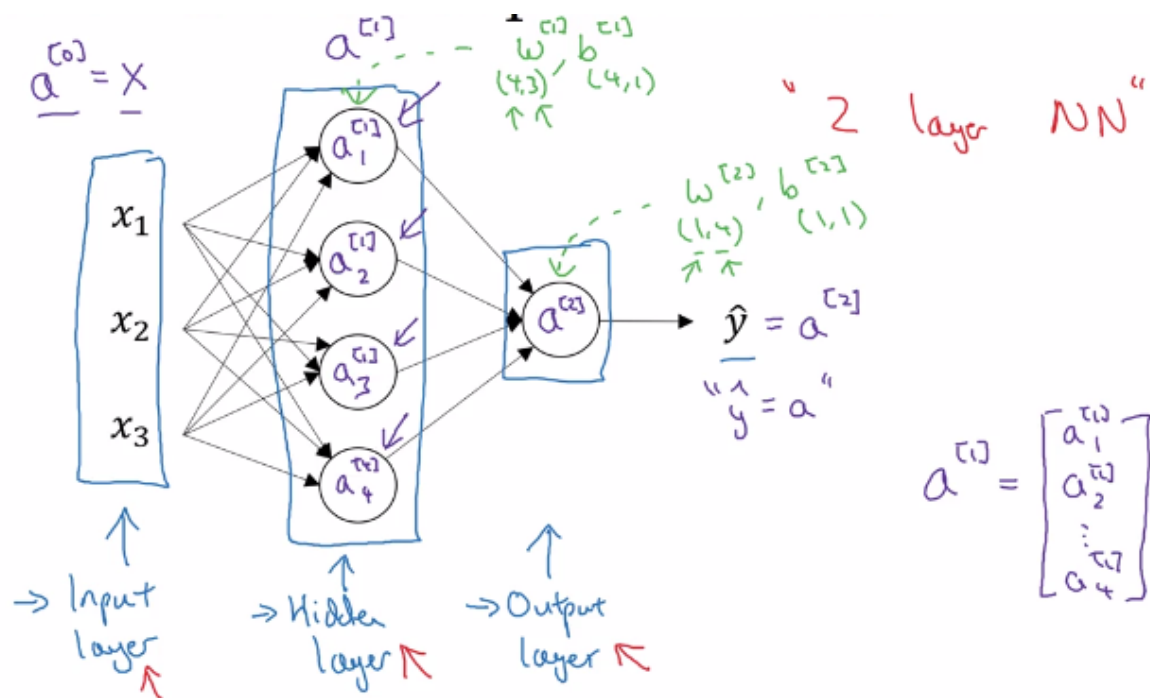
```
a = np.random.randn(5, 1) → a.shape = (5, 1) column vector ✓
```

```
a = np.random.randn(1, 5) → a.shape = (1, 5) row vector ✓
```

Week 3: Shallow Neural Network

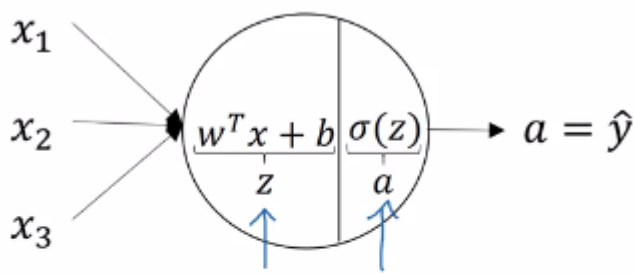
Representación de Redes Neuronales

Una red neuronal con una capa oculta.



¿Cómo se computan las salidas?

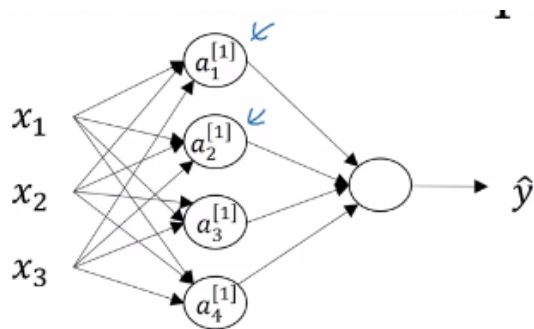
Un nodo en una red neuronal consiste en dos pasos de una regresión logística.



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

$$a = \sigma(z)$$

Una red neuronal hace esto mismo, pero más veces. Esto es



$$z_1^{[1]} = \underline{w_1^{[1]T}} x + b_1^{[1]}, a_1^{[1]} = \sigma(z_1^{[1]})$$

$$z_2^{[1]} = w_2^{[1]T} x + b_2^{[1]}, a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3^{[1]}, a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4^{[1]}, a_4^{[1]} = \sigma(z_4^{[1]})$$

donde la notación viene dada por

$\begin{cases} z_i \leftarrow \text{layer} \\ a_i \leftarrow \text{node in layer.} \end{cases}$

Veamos cómo hacer para vectorizar estas ecuaciones

$$z^{[1]} = \begin{bmatrix} -w_1^{[1]T} \\ -w_2^{[1]T} \\ -w_3^{[1]T} \\ -w_4^{[1]T} \end{bmatrix}_{(4,3)} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} \rightarrow w_1^{[1]T} x + b_1^{[1]} \\ \rightarrow w_2^{[1]T} x + b_2^{[1]} \\ \rightarrow w_3^{[1]T} x + b_3^{[1]} \\ \rightarrow w_4^{[1]T} x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix}$$

Given input x:

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

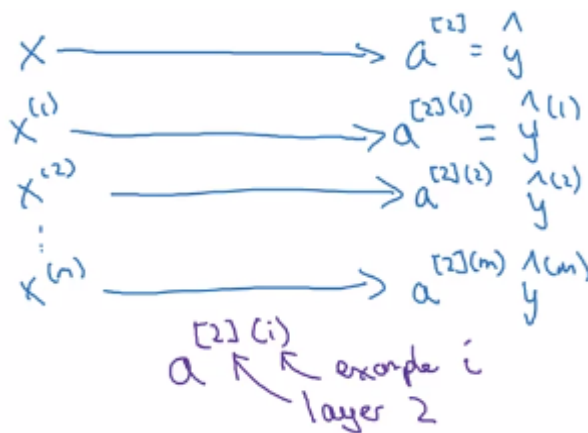
$$\rightarrow \underset{(4,1)}{a^{[1]}} = \sigma(\underset{(4,1)}{z^{[1]}})$$

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

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

Veamos ahora cómo predecir la salida de una red neuronal pero dadas varias entradas, no solo un ejemplo.

Introduciendo un poco de notación



Loopeando sobre todos los ejemplos

for $i = 1$ to m :

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

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

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

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

Lo que queremos hacer ahora es vectorizarlo. Escribiendo las entradas en columnas,

← training samples
↑ hidden units.

La vectorización resulta (Forward propagation)

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix} \leftarrow$$

$$\underline{A^{[1]}} = \begin{bmatrix} | & | & & | \\ a^{1} & a^{[1](2)} & \dots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

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

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

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

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

Explicación de la implementación vectorizada

$$z^{[1]} = W^{[1]}x + b^{[1]}, \quad z^{[2]} = W^{[2]}x^{[1]} + b^{[2]}, \quad z^{[3]} = W^{[3]}x^{[2]} + b^{[3]}$$

$$W^{[1]} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}, \quad W^{[1]}x^{(1)} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}, \quad W^{[1]}x^{(2)} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}, \quad W^{[1]}x^{(3)} = \begin{bmatrix} \vdots \\ \vdots \\ \vdots \end{bmatrix}$$

$$W^{[1]} \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} | & | & | \\ z^{[1]}(1) & z^{[1]}(2) & z^{[1]}(3) \\ | & | & | \end{bmatrix} = z^{[1]}$$

$X \quad \quad \quad W^{[1]}x^{(1)} = z^{[1]}(1)$

La función sigmoide no es la mejor elección en redes neuronales. Estudiemos otras funciones de activación más útiles.

Funciones de activación

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

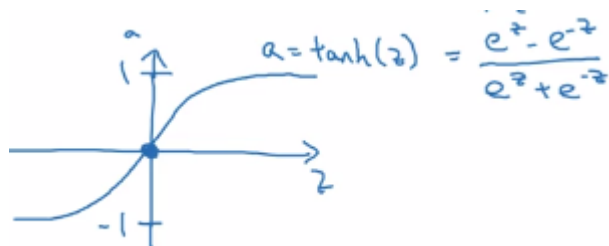
$$a^{[1]} = \cancel{\sigma(z^{[1]})} \quad g(z^{[1]})$$

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

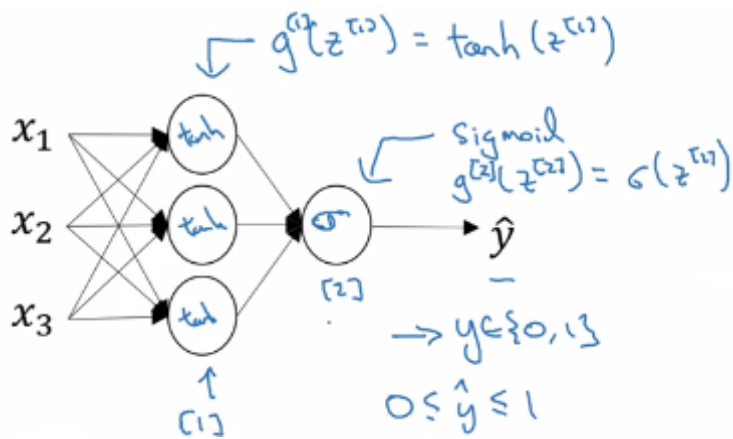
$$a^{[2]} = \cancel{\sigma(z^{[2]})} \quad g(z^{[2]})$$

Reemplacemos la función sigmoide por una función de activación general $g()$.

En general la tangente hiperbólica funciona mejor.

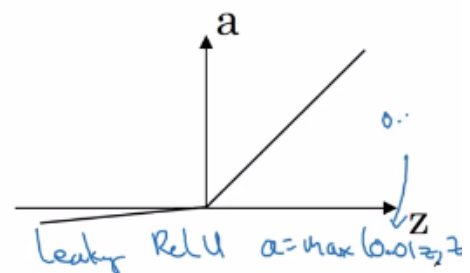
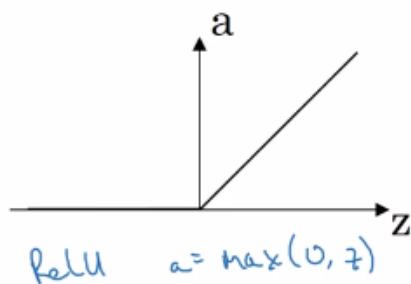
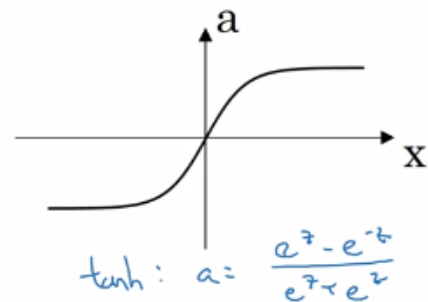
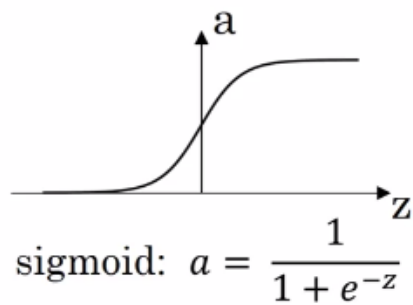


Pero se pueden utilizar distintas funciones de activación en distintas capas. Es conveniente utilizar en la salida la función sigmoide para clasificación binaria, ya que entrega valores entre 0 y 1. Pero en las capas ocultas conviene utilizar la tangente hiperbólica.



Ventajas y desventajas de las funciones de activación

En resumen,



Derivadas de las funciones de activación

Para backpropagation necesitamos conocer las derivadas de las funciones de activación.

- Función de activación sigmoide

$$\begin{aligned}\frac{d}{dz} g(z) &= \text{slope of } g(z) \text{ at } z \\ &= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right) \\ &= g(z) (1 - g(z))\end{aligned}$$

- Función de activación tangente hiperbólica

$$\begin{aligned}g'(z) &= \frac{d}{dz} g(z) = \text{slope of } g(z) \text{ at } z \\ &= 1 - (\tanh(z))^2\end{aligned}$$

- RELU

$$\begin{aligned}g(z) &= \max(0, z) \\ g'(z) &= \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \\ &\quad \text{~~undefined if } z = 0~~\end{aligned}$$

- Leaky RELU

$$\begin{aligned}g(z) &= \max(0.01z, z) \\ g'(z) &= \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}\end{aligned}$$

Gradiente descendente para redes neuronales

Consideremos primero una capa oculta nomás.

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$
 $(n^{[1]}, n^{[2]})$ $(n^{[2]}, 1)$ $(n^{[1]}, n^{[2]})$ $(n^{[2]}, 1)$

$$n_x = n^{[0]}, \quad n^{[1]}, \quad \underline{n^{[2]} = 1}$$

Previamente habíamos visto como calcular las predicciones \hat{y} , ahora las derivadas de la función de coste.

$$\text{Cost function: } J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^m \ell(\hat{y}_i, y_i)$$

\uparrow
 $a^{[2]}$

Gradient descent:

→ Repeat {

 Compute predicts $(\hat{y}^{(i)}, i=1, \dots, m)$

$dW^{[1]} = \frac{\partial J}{\partial W^{[1]}}$, $db^{[1]} = \frac{\partial J}{\partial b^{[1]}}$, ...

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

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

Fórmulas para calcular derivadas (Backward propagation)

Back propagation:

$dZ^{[2]} = A^{[2]} - Y \leftarrow Y = [y^{(1)}, y^{(2)}, \dots, y^{(m)}]$

$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$ $(n^{[1]}) \leftarrow$

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

$dZ^{[1]} = \underbrace{W^{[2]T}}_{(n^{[2]}, m)} dZ^{[2]} \otimes \underbrace{g^{[1]'}(Z^{[1]})}_{\text{element-wise product}} \quad (n^{[1]}, m)$

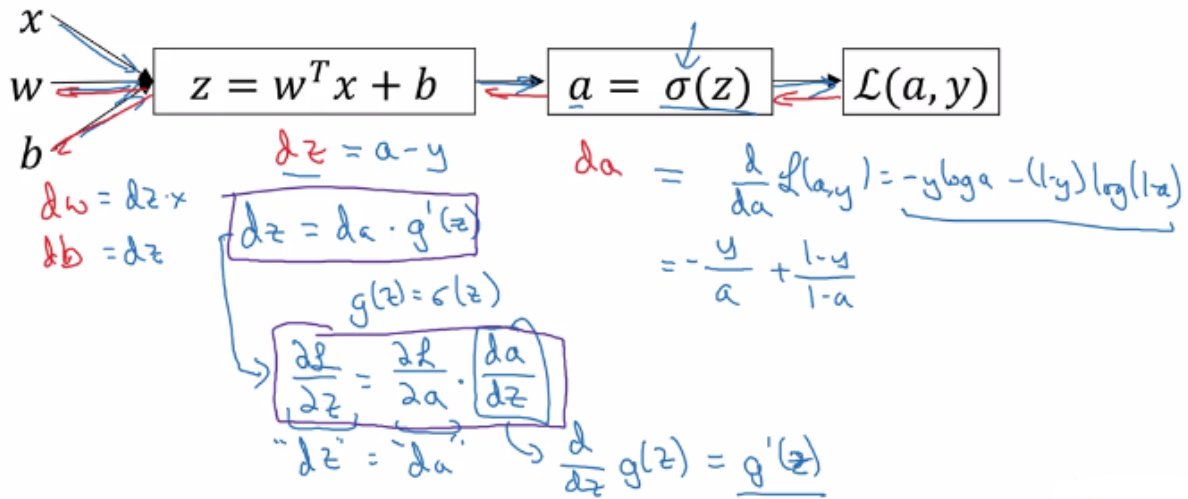
$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$

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

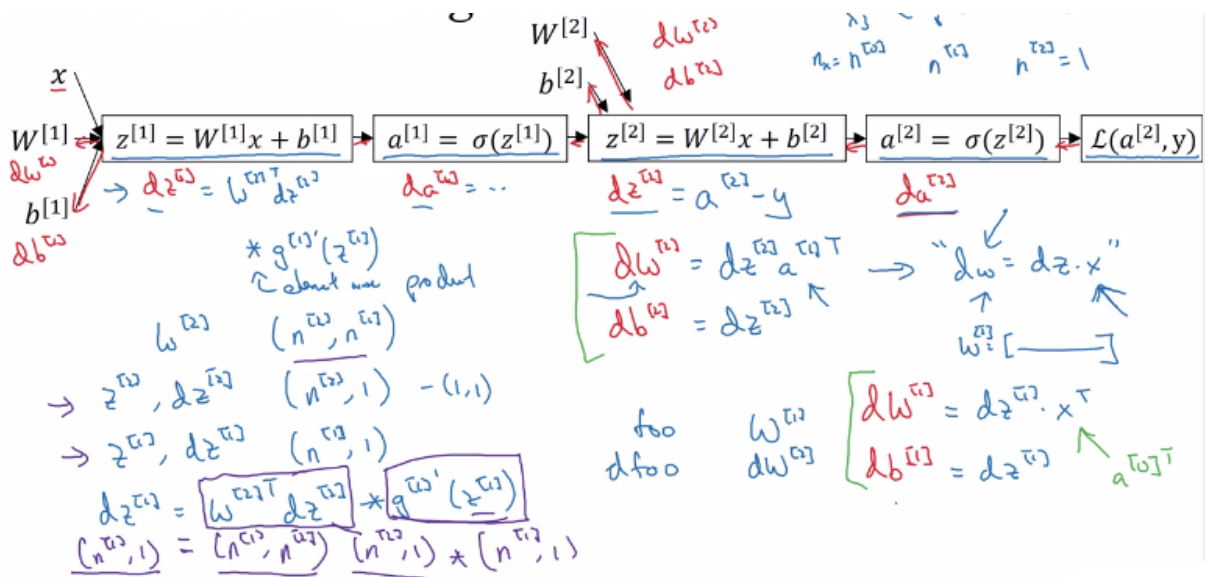
$(n^{[1]}, 1)$ $(n^{[1]}, 1)$

Intuición de la backward propagation

- Regresión logística



- Gradiente en red neuronal



De manera más prolija, las 6 ecuaciones de interés para el backpropagation

$$\underline{dz^{[2]}} = \underline{a^{[2]}} - \underline{y}$$

$$dW^{[2]} = dz^{[2]} a^{[1]T}$$

$$db^{[2]} = dz^{[2]}$$

$$\underset{(n^{[1]}, 1)}{dz^{[1]}} = W^{[2]T} dz^{[2]} * \underset{(n^{[1]}, 1)}{g^{[1]}'(z^{[1]})}$$

$$dW^{[1]} = dz^{[1]} x^T$$

$$db^{[1]} = dz^{[1]}$$

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

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

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

$$\underset{(n^{[2]}, m)}{dZ^{[1]}} = \underset{(n^{[2]}, m)}{W^{[2]T} dZ^{[2]}} * \underset{(n^{[2]}, m)}{g^{[1]}'(Z^{[1]})}$$

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

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

$$J(\dots) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}_i, y_i)$$

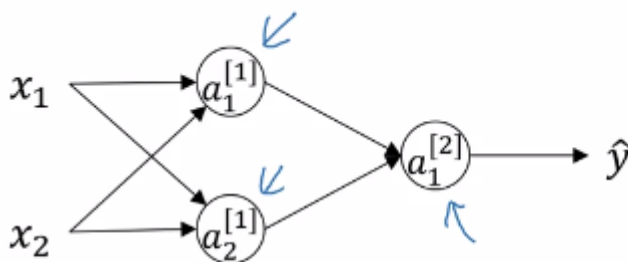
elementwise product

Inicialización parámetros

- En regresión logística se puede inicializar los parámetros con valor cero.
- Pero la red neuronal hay que inicializarla aleatoriamente. Aplicar gradiente descendente a una red inicializada con ceros, no funcionará.

Inicialización aleatoria

El factor 0.01 en w es para que se inicialicen con valores pequeños. Si tomasen valores grandes la función de activación saturaría. Por otra parte, a los parámetros b no les afecta inicializarlos con valores nulos.



$$\begin{aligned} \rightarrow w^{[1]} &= np.random.randn(2,2) * \frac{0.01}{100?} \\ b^{[1]} &= np.zeros(2,1) \\ w^{[2]} &= \dots \\ b^{[2]} &= 0 \end{aligned}$$

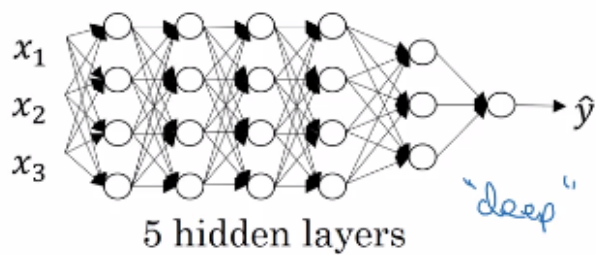
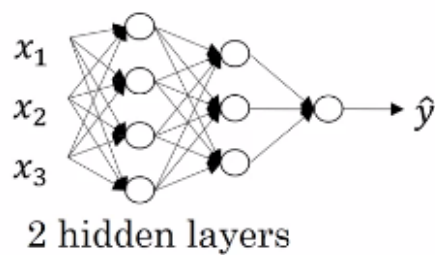
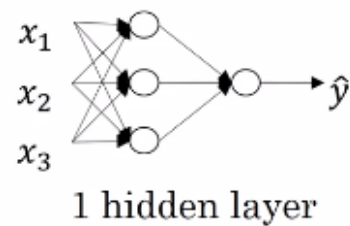
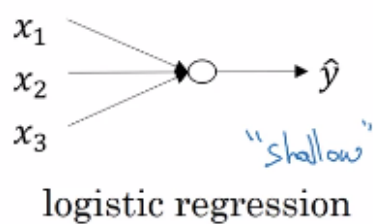


Cuando se entrene alguna red neuronal más profunda sería más razonable usar otro valor que no fuera 0.01, pero se verá más adelante.

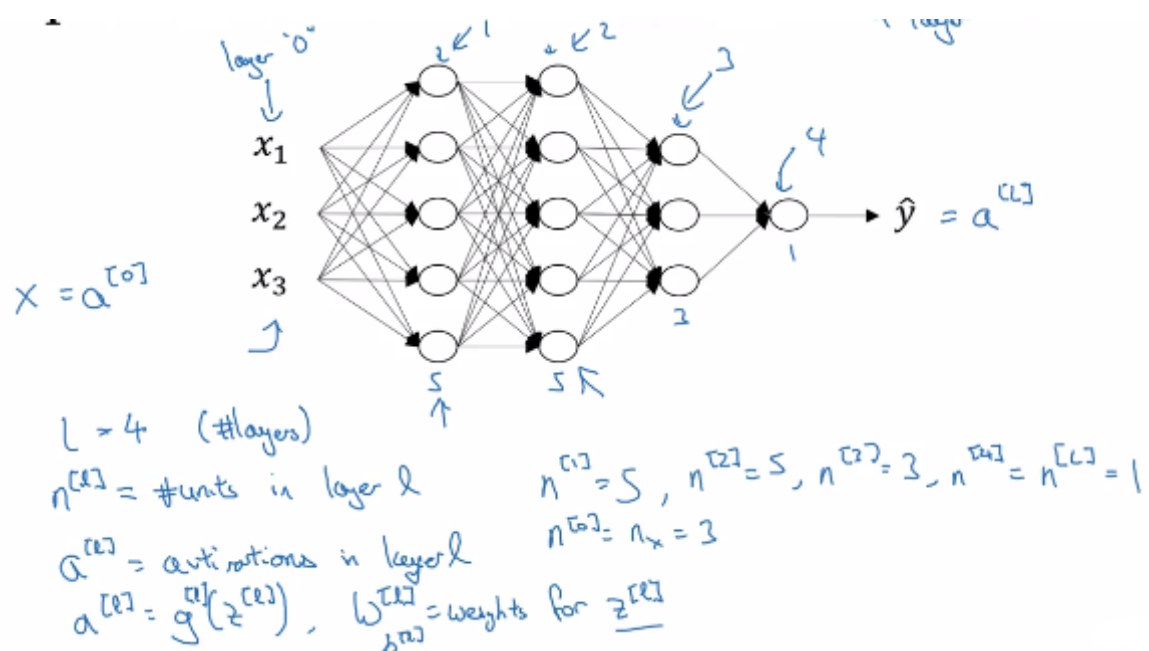
Week 4 Deep Neural Network

Redes neuronales profundas de L-capas

Ejemplos de redes neuronales.



Veamos la notación a utilizar



Forward and Backward propagation in deep neural networks

- Forward propagation

Input $a^{[l-1]}$ ←

Output $a^{[l]}$, cache $(z^{[l]})$ ←

$$z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]}$$

$$a^{[l]} = g^{[l]}(z^{[l]})$$

Vectorized:

$$z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$$

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

Estas dos ecuaciones se encuentran en un for-loop, el cual no parecería que se puede vectorizar.

- Backward propagation

Input $da^{[l]}$

Output $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

$$dz^{[l]} = da^{[l]} \otimes g^{[l]'}(z^{[l]})$$

$$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$$

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

$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

$$dz^{[l-1]} = W^{[l+1]T} \cdot dz^{[l]} \otimes g^{[l+1]'}(z^{[l+1]})$$

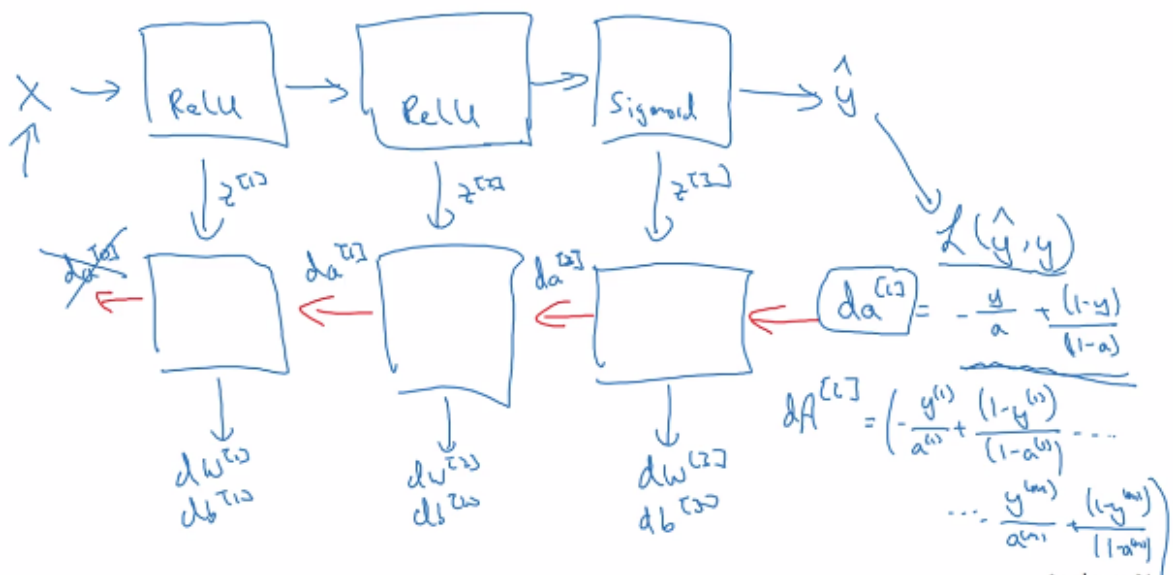
$$dz^{[l]} = dA^{[l]} \otimes g^{[l]'}(z^{[l]})$$

$$dW^{[l]} = \frac{1}{n} dz^{[l]} \cdot A^{[l-1]T}$$

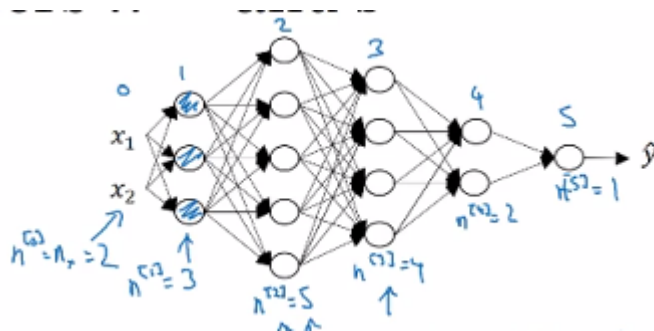
$$db^{[l]} = \frac{1}{n} \text{np.sum}(dz^{[l]}, \text{axis}=1, \text{keepdims}=True)$$

$$dA^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

Resumen



Método de debugging: dimensiones de las matrices



Los parámetros w para un ejemplo:

$$W^{[1]}: (n^{[1]}, n^{[0]})$$

$$W^{[2]}: (5, 3) \quad (n^{[2]}, n^{[1]})$$

De manera general.

$$W^{[L]}: (n^{[L]}, n^{[L-1]})$$

Para b

$$b^{[L]}: (n^{[L]}, 1)$$

Para backpropagation dw y db tienen la misma dimensión que W y b respectivamente.

$$W^{(1)}: (n^{(1)}, n^{(0)})$$

$$b^{(1)}: (n^{(1)}, 1)$$

Otras cantidades de interés: $z = g(a)$, y a tienen la misma dimensión,

Considerando los m ejemplos a entrenar

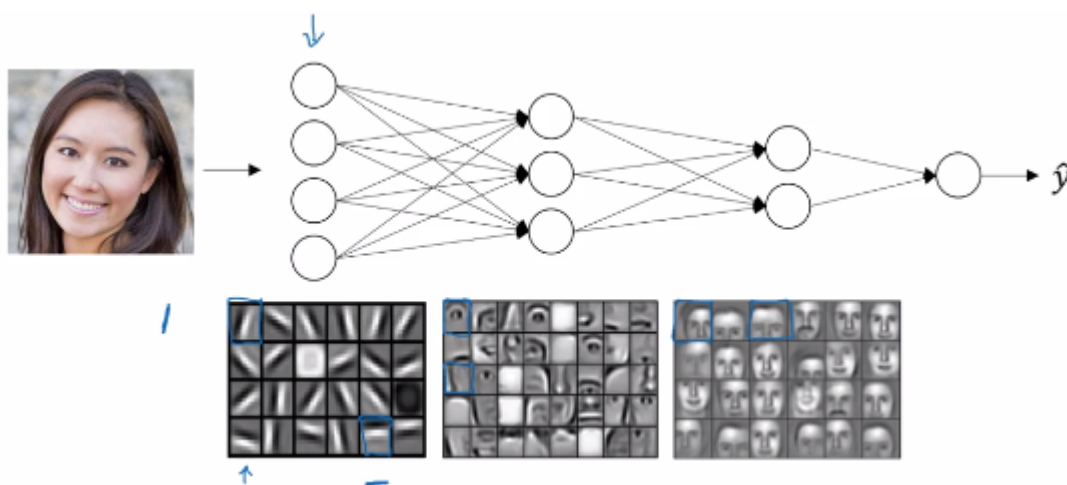
$$\vec{z}^{(1)} = W^{(1)} \cdot X + b^{(1)}$$

$(n^{(1)}, m)$ $(n^{(1)}, n)$ $(n^{(1)}, m)$ $(n^{(1)}, 1)$
 $(n^{(0)}, m)$

En Python b podría seguir siendo de la misma dimensión, pero por Broadcasting es redimensionada al tamaño correcto.

¿Por qué las redes neuronales profundas generalmente funcionan mejor que las que no lo son?

Como ejemplo en la identificación de una cara. Se podría pensar a la primera capa de la red neuronal como la encargada de identificar bordes. A la siguiente para identificar ojos, o narices mediante el encajado de bordes. Mediante el rejunte de ojos, narices la red puede detectar caras.



Para un audio, a bajo nivel la red neuronal podría detectar los cambios de sonido, luego unidades básicas de sonido, como fonemas, luego poder reconocer palabras, para luego reconocer oraciones o frases.



¿Qué son los hiper parámetros?

Dados los parámetros de una red neuronal,

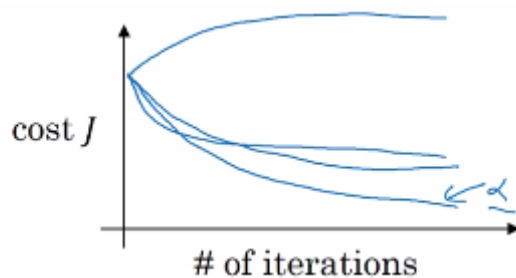
Parameters: $W^{[1]}$, $b^{[1]}$, $W^{[2]}$, $b^{[2]}$, $W^{[3]}$, $b^{[3]}$...

los hiper parámetros son aquellos que controlan y determinar el valor de los parámetros.

Ellos son:

- Learning ratio alfa
- número de iteraciones
- número de capas ocultas L
- número de unidades ocultas $n^{[1]}, n^{[2]}, \dots$
- elección de funciones de activación
- etc.

Deep Learning es un proceso empírico. El valor de los hiper parámetros se calcula mediante la experimentación.



Forward y Backward propagation

$Z^{[1]} = W^{[1]}X + b^{[1]}$ $A^{[1]} = g^{[1]}(Z^{[1]})$ $Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$ $A^{[2]} = g^{[2]}(Z^{[2]})$ \vdots $A^{[L]} = g^{[L]}(Z^{[L]}) = \hat{Y}$	$dZ^{[L]} = A^{[L]} - Y$ $dW^{[L]} = \frac{1}{m} dZ^{[L]} A^{[L]T}$ $db^{[L]} = \frac{1}{m} np.sum(dZ^{[L]}, axis = 1, keepdims = True)$ $dZ^{[L-1]} = dW^{[L]T} dZ^{[L]} g'^{[L]}(Z^{[L-1]})$ \vdots $dZ^{[1]} = dW^{[L]T} dZ^{[2]} g'^{[1]}(Z^{[1]})$ $dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[1]T}$ $db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$
---	---

