REDES NEURONALES **4 de marzo de 2019**

4 de marzo de 2019

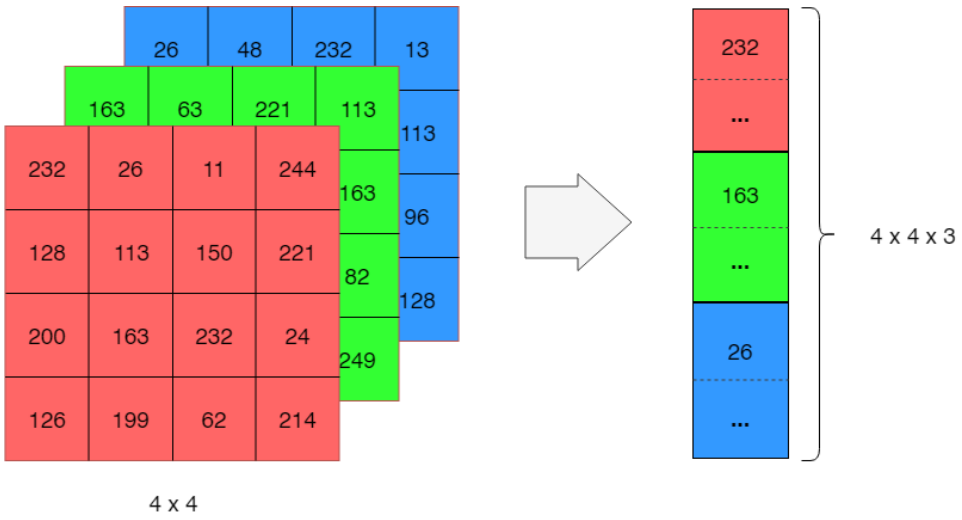
Índice

1. Regresión Logística

1.1. Descripción del problema

Regresión logística es un algoritmo de clasificación binaria. Dada una imagen, queremos mostrar si pertenece a una categoría o no. La imagen está formada por H x W píxeles y normalmente es de tipo RGB, representando los distintos colores. En esencia, **las imágenes son 3 matrices H X W, cada una representando un color**.

Para facilitar el trabajo, una imagen se representará mediante un vector, en la que primero estén los H x W valores para el rojo, después para el verde y finalmente para el azul. Teniendo el vector final una dimensión de **H x W x 3**, a lo que denominaremos como Nx. Este proceso es el de normalizar la imagen.



Normalización de imagen.

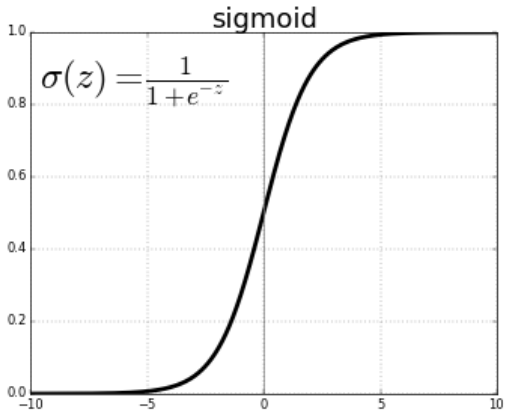
En el problema, tendremos un número de imágenes **m** de entrada para entrenar o testear nuestra red neuronal. Por lo que, normalizando esas imágenes, nos quedaría una matriz de dimensión Nx x m, denominada como **X** Cabe añadir que también se necesita una estructura de datos que represente la categoría a la que pertenece una imagen. En éste caso estará formado por 1's representando que la imagen es un gato, y 0's representando que no. La estructura de datos que se usará será una matriz de **1 x m** columnas, denominada como **Y**.

En resumen, los datos de entrada serán pares **(X(1), Y(1)), ..., (X(m), Y(m))**.

1.2. Objetivo

El objetivo del algoritmo es que, dada una imagen de entrada X (en la forma previamente expuesta), éste devuelva ŷ, la probabilidad de que esa imagen pertenezca a la categoría. A su vez, se usarán dos parámetros: - **w:** Vector de tamaño Nx, al igual que X. - **b:** Número real.

Para estimar la categoría, se podría utilizar una función linear con la entrada X, tal que: $z = w^T x^{(i)} + b$. No obstante, necesitamos que la probabilidad se encuentre entre 0 y 1, por lo que utilizaremos la función sigmoide. A esta le llamaremos a (activación) y su valor será $\hat{y} = a = sigmoid(w^T x^{(i)} + b)$. Véase en la siguiente gráfica.



Función sigmoide.

Lo que se quiere conseguir es que dada la entrada **(X(1), Y(1)), ..., (X(m), Y(m))**, $\hat{y} \approx y$, para cada x.

1.3. Entrenamiento

- **Función de Loss o Error (L):** Se encarga de medir los aciertos de predicción del algoritmo para un único ejemplo. Cabe añadir que $a = \hat{y}$. Se utiliza la función:

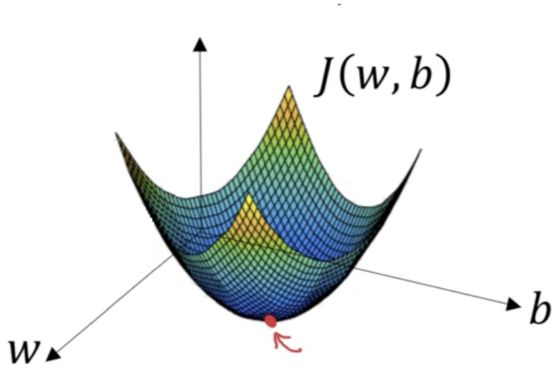
$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

  Esta función nos permite que, cuando y es 1, ŷ tomé un valor muy grande. Por el contrario, cuando y es 0, $\hat{y}$ tomará un valor muy pequeño.

- **Función de coste (J):** Se encarga de medir los aciertos de predicción del algoritmo en el conjunto de entrenamiento completo. Se trata de la media de las funciones de loss para todas las imagenes de entrada.

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

- **Descenso por gradiente:** Algoritmo para entrenar o aprender los parámetros w y b del conjunto de entrenamiento. La función de coste J es una función convexa, dónde J(w, b) es la altura. Queremos hallar el punto en esa malla convexa donde J(w, b) sea mínimo. En cada iteración del algoritmo, se modifican los valores de w y b para que el coste sea menor.

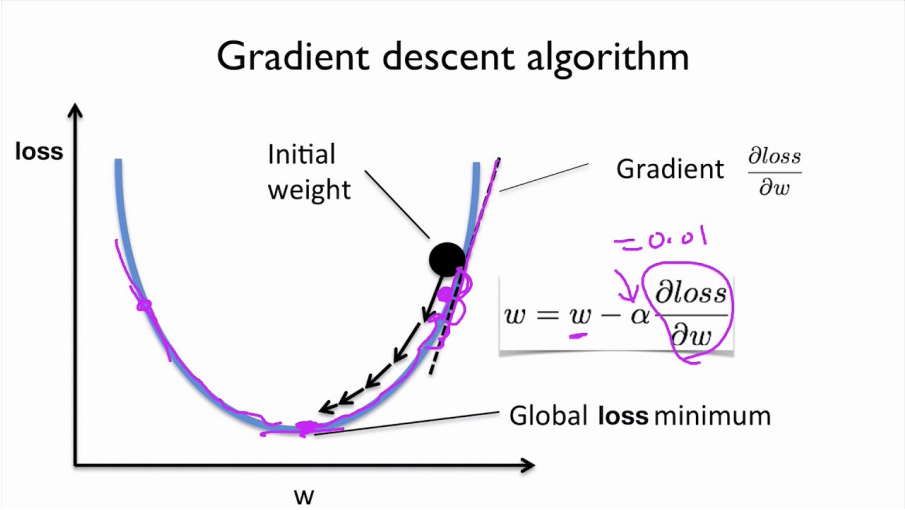


Descenso por gradiente.

Para modificar los valores de w y b, de manera que siempre se consiga un coste menor, se utilizará la pendiente de la función de coste respecto a ese valor, es decir, la derivada de J respecto a w: $\frac{\partial J}{\partial w}$. Para simplificarlo, siempre que se trate de una derivada parcial respecto a la variable objetivo J, se representara como: dw.

Al tratarse de una función convexa, la pendiente será positiva cuando el valor de w sea grande y negativa cuando sea pequeño. De ésta manera nos aseguramos que el valor de la variable siempre se dirige al óptimo global. La operación a realizar para mejorar la variable sería:

$$\theta = \theta - \alpha d\theta$$



Descenso por gradiente.

Cabe añadir que $d\theta i$, no sólo se refiere a la derivada parcial de θ respecto a J en una imagen i. Se trata de una variable que se obtiene al realizar ésta computación por cada imagen, pues θ será distinta en cada una. Finalmente se haría la media de la acumulación de $d\theta i$, lo que correspondería a $d\theta$ en la formula expuesta previamente. Por último esto se lleva a cabo para todas las variables que afecten a la función de coste J.

1.4. Codificación en Python

1.4.1. Paquetes

- numpy es un paquete fundamental para computación científica con Python.
- h5py es un paquete común para interactuar con un dataset guardado en un fichero H5.
- matplotlib se trata de una famosa librería para representar gráficos en Python.
- PIL y scipy se usan para testear el modelo con una figura personalizada.

1.4.2. Datos a usar

```
# Ense ando una imagen
plt.imshow(train_set_x_orig[picture_index])
print("Label = " + classes[np.squeeze(train_set_y_orig[picture_index])].
        decode("utf-8"))

# Tama o de la imagen
print (train_set_x_orig[picture_index].shape)
# (50, 50, 3) Tres matrices de cada color, cada una compuesta por 50 x 50 p xeles.
print (train_set_x_orig.shape)
# (120, 50, 50, 3) 120 im genes de entrenamiento.

# Normalizar las im genes (num_px, num_px, 3) a vectores de dimensi n (num_px     num
# Truco para obtener (num_px     num_px     3, train_set_len) -> (shape[0], -1)
train_set_x_flatten = train_set_x_orig.
        reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.
        reshape(test_set_x_orig.shape[0], -1).T

# Estandarizar las im genes - Sustraer la media de cada ejemplo.
train_set_x = train_set_x_flatten / 255
test_set_x = test_set_x_flatten / 255
```

1.4.3. Vectorización

El producto escalar (Dot Product) se trata de la multiplicación entre dos vectores que resulta en un número escalar. Para ello, si se realiza el producto escalar entre a y b, se multiplicará a por la traspuesta de b, resultando en un número. Esto lo necesitamos para calcular $z = w^T x^{(i)} + b$. En Python, se podría hacer mediante dos maneras:

- **Tradicional:** Usando un bucle for en el que dentro tengamos un acumulador: z[i] += w[i] * x[i]
- **Vectorizado:** Usando z[i] = np.dot(w, x[i]) + b, de la librería numpy. Mucho mas eficiente, al utilizar instrucciones SIMD (Single Instruction Multiple Data), pudiéndose usar en CPU y GPU (mas rápido).

A su vez, dado que lo único que varía en cada z[i] es x[i], se podría realizar directamente el producto escalar entre w y X sumado con b. resultando en un vector con los resultados de

productos escalares: $Z = w^T X + b = [w^T x^{(1)} + b,\ w^T x^{(2)} + b,\ ...,\ w^T x^{(m)} + b]$. Esto se puede hacer con Z = np.dot(w.T, X) + b, gracias a **broadcasting** de Python.

El siguiente paso sería calcular la **sigmoide** en $\hat{y} = a = sigmoid(w^T x + b)$. Recordemos que la fórmula de la función sigmoide es: $sigmoid(w^T x + b) = \frac{1}{1+e^{-(w^T x+b)}}$. Esto se podría hacer de manera eficiente con s = 1 / (1 + np.exp(-z)). Al utilizar np.exp() y no math.exp(), se puede recibir un numpy array o escalar y la función se aplicará en cada elemento del array o simplemente en el escalar, de manera automática.

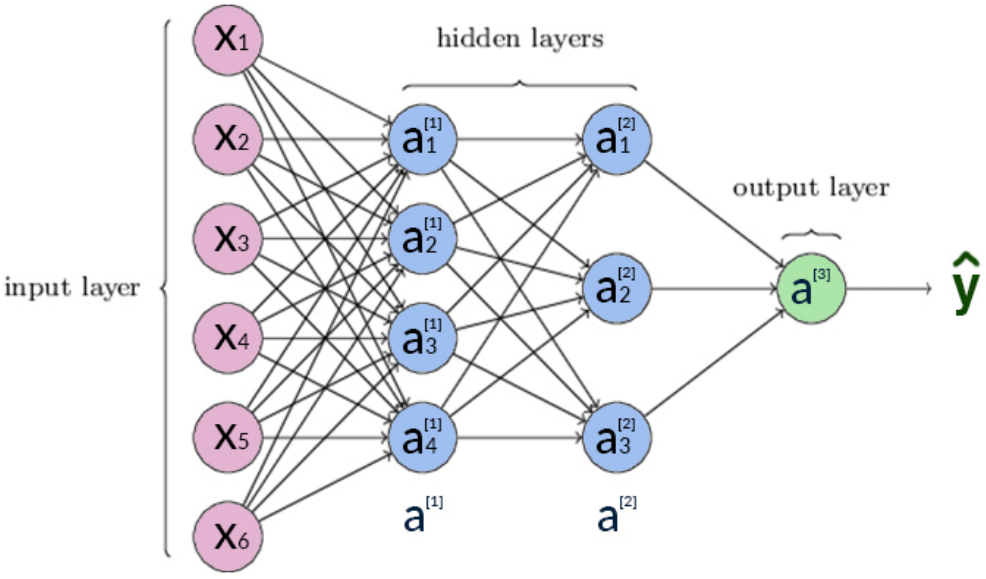
2. Redes Neuronales

2.1. Representación

Una Red Neuronal está compuesta por varias capas. Se pueden encontrar tres tipos de capas:

- **0- Capa de Entrada:** Contiene las entradas de la red neuronal, en esencia un conjunto de características.
- **1- Capa(s) Oculta(s):** Se considera oculta porque desde el conjunto de entrenamiento no se pueden ver los valores, a diferencia de las otras dos capas.
- **2- Capa de Salida:** Responsable de general el valor predicho $\hat{y}$.

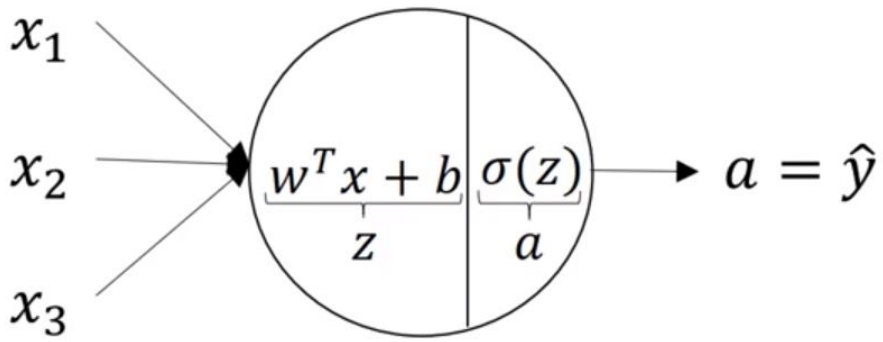
La forma mas básica en una red neuronal es de 2 capas (2-layer NN), pues la capa de entrada no se considera una capa oficial. A su vez, cada capa está compuesta por varias neuronas o unidades, que generan valores de activación, formando una matriz columna con los distintos valores, denominada **a**. Según la posición en la que está la capa, esa matriz de activación se representa como $a^{[0]}$ (capa de entrada), $a^{[1]}$ (primera capa oculta)...



3-layer NN.

El comportamiento de una Red Neuronal es el mismo que repetir regresión logística varias veces. Dentro de cada neurona, se calcula el valor de activación a partir de aplicar una función

de activación a una función linear. En ésta función linear intervienen dos parámetros w.T y b. Estos parámetros se inicializarán de manera aleatoria, pues si se inicializan todos a 0 en una misma capa, el comportamiento y los valores de activación generados serán iguales.



Cómputos que lleva a cabo una neurona en la capa 1.

En la figura representando una neurona, se ha expuesto los distintos cómputos que una neurona debe realizar en la primera capa. Por ello, ésta recibe X como entrada, que será multiplicada por w.T. No obstante, si estuviésemos en la segunda capa, ésta computación cambiaría, pues ya no se recibe X, si no las salidas de los valores de activación de la capa previa, es decir, la capa i recibiría la matriz columna $a^{[i-1]}$. Quedando las computaciones de una neurona de la capa dos tal que:

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

En resumen, una Red Neuronal, dada un conjunto de características X como entrada, genera un valor final de activación $\hat{y}$. No obstante, para entrenar la Red Neuronal, se utilizarán varios ejemplos, por lo que necesitaremos calcular $\hat{y}$ para cada entrada X[i]. Para poder calcular esto, la manera común sería hacer un bucle que recorra todas las entradas. Sin embargo, es un mecanismo muy ineficiente que se podría reemplazar a través de la vectorización.

Recordemos que W[i] y X[i] era una matriz columna de dimensiones (Nx x 1) y b[i] era un valor real. Para poder tener en cuenta varias entradas a la vez, se sustituirán W[i] y X[i] por matrices de dimensiones (Nx x m), de manera que cada columna corresponda a una entrada distinta. Y b[i] por un vector de valores reales, con el mismo significado. Por lo tanto, z[i] y a[i] corresponderían también a una matriz (Nx x m).

Tras realizar éstos cambios, la salida de la Red Neuronal también cambiaría, en vez de ser el valor de predicción $\hat{y}$, será un conjunto de predicciones $\hat{y}[i]$, cada uno asociado a una entrada distinta.

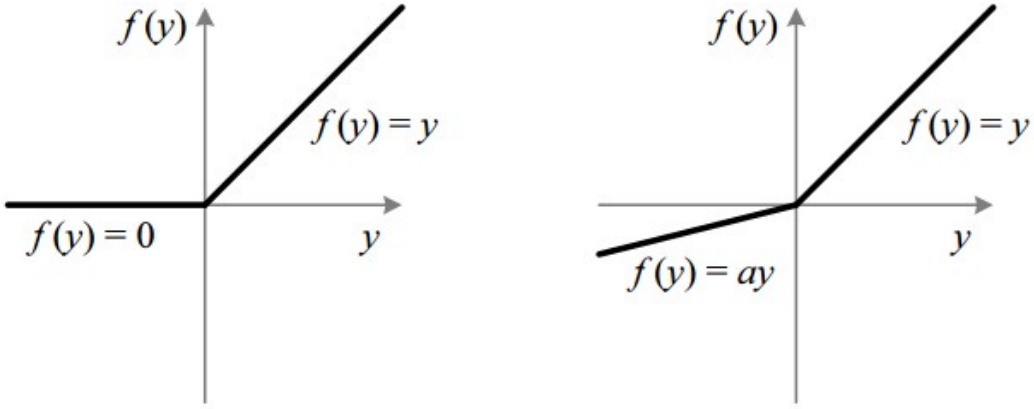
2.2. Funciones de activación

1. **Función Sigmoide:** Función elemental. Sólo se suele usar en la última capa si se debe devolver como resultado de la Red Neuronal una probabilidad.

2. **Función Tanh:** Superior que la función sigmoide. Los valores de y varían entre 1 y -1, haciendo que la media se encuentre mas cerca de 0 que en la función sigmoide.

3. **Función RELU (Recitifed Line Unit):** La opción por defecto para una función de activación. Corrige el error de Tanh o Sigmoide en las que si los valores son muy grandes, la pendiente será 0 y el proceso de aprendizaje se verá ralentizado. En éste caso si z es menor que 0 la pendiente es 0 y si no 1.

4. **Función Leaky RELU:** Similar a la función RELU con la diferencia de que si z es menor que 0 la pendiente puede ser mayor que 0.



Funciones RELU y Leaky RELU.