**Reporte de práctica 1**

Para realizar está práctica se hizo uso del lenguaje Python, un poderoso lenguaje de programación orientado a objetos de alto nivel que tiene múltiples frameworks y librerías que nos permiten hacer machine learning de una forma más sencilla, como lo es el caso de tensorflow, keras, numpy o pythorch, de esta forma podemos elegir varias formas de implementación, en este caso, se opto por la siguiente librería:

**-Numpy**

Una librería bastante optimizada que permite la operación entre matrices y vectores de una forma mucho más rápida que si lo hacemos de manera convencional, por lo tanto, y para una técnica como lo es backpropagation nos permite tener un tiempo de ejecución y entrenamiento más rápido, aunque, evidentemente esto depende del problema a resolver, en este caso, nuestro problema a resolver es bastante sencillo, al tener que entrenar una red que resuelva de una a cinco compuertas al mismo tiempo, las cuales son AND, OR, NAND, NOR, y XOR.

1. Inicializar los pesos y bias de forma aleatoria, usando una distribución aleatoria gaussiana normalizada.

Para realizar este primer punto se uso una función de la librería numpy, llamada:

**random.normal(size=(rows,cols))**

Donde:

rows = número de filas de la matriz

cols = número de columnas de la matriz

1. Modificar el programa anterior con el objetivo de poder variar el número de neuronas en la capa oculta a petición del usuario.

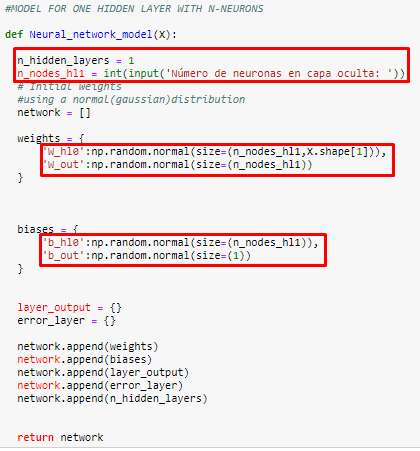
Para realizar esto se desarrollo una función que permitía al usuario ingresar el número de neuronas de la capa oculta, con este valor era posible inicializar los pesos y los bias de dicha capa, dado que, la dimensión de la matriz de pesos y de la matriz de bias de la capa oculta depende directamente del número de neuronas, donde podemos observar que:

**rows\_matrix = número de neuronas de la capa oculta.**

**cols\_matrix = número de neuronas o entradas de la capa anterior.**

Por lo tanto si queremos 10 neuronas en la capa oculta, y tenemos una entrada de 5 vectores, la dimensión de la matriz de pesos será **dim\_matrix = (10,5)** de modo que el número de pesos en la matriz es de **10\*5=50,** pues cada neurona tendrá 5 pesos sinápticos.

A continuación se muestra la imagen donde se inicializan los pesos y bias que dependen del número de neuronas de la capa oculta.



En donde observamos en los recuadros de color rojo los pasos que se siguieron para inicializar los pesos y bias con base al número de neuronas del usuario, siendo:

**n\_nodes:** Número de neuronas en la capa oculta.

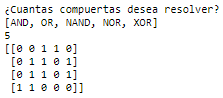
**X.shape[1]:** Con X siendo la matriz que contiene el conjunto de vectores de entrenamiento, lo que hacemos es obtener la forma en la que está distribuida esa matriz, es decir, número de filas y columnas, con el atributo shape podemos obtener una tupla de la siguiente manera **(n\_rows,n\_cols)**, en este caso n\_rows corresponde al número de vectores de entrenamiento que vamos a ingresar a la red, mientras que n\_cols nos indica el tamaño de cada uno de esos vectores, por lo tanto obtenemos el segundo valor(n\_cols) de la forma **shape[1].**

1. Modificar el programa anterior con el objetivo de poder variar el número de neuronas en la capa de salida a petición del usuario; con éste se podrán resolver m cantidades de compuertas lógicas simultáneamente.

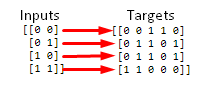
En este punto había que modificar los targets correspondientes a nuestros vectores de entrenamiento, es decir, el número de salidas que se deben obtener ya no será el correspondiente a una sola neurona, lo será para cada compuerta que deseemos resolver, en este caso, se resolverán hasta un máximo de 5 compuertas lógicas, ya antes mencionadas, por lo tanto, al usuario se le debe preguntar cuántas compuertas lógicas pretende resolver, para poder inicializar los targets correspondientes a dicho número elegido por el usuario, para este caso, el formato de salida de las compuertas elegido es:



Como podemos ver la primer neurona de salida de la red corresponderá a la salida de la compuerta AND, la segunda a la OR, y así sucesivamente, de modo que si el usuario decide resolver tres compuertas, el vector estará conformado por [AND,OR,NAND], en la siguiente imagen el usuario decidió resolver las cinco compuertas lógicas:

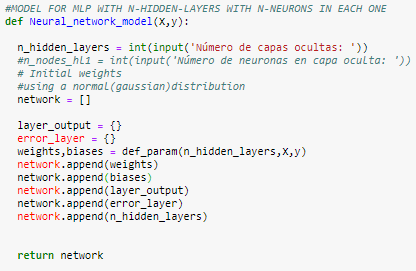


Por lo tanto, los targets para cada entrada serían:



1. Modificar el programa anterior con el objetivo de poder variar el número de capas ocultas. Cada capa oculta puede variar el número de neuronas en cada capa y la capa de salida puede resolver m cantidades de compuertas lógicas simultáneamente.

Para este punto se editó la anterior función de la arquitectura de la red neuronal, de modo que podamos decidir cuántas capas ocultas se desean y además el número de neuronas en cada una para su inicialización, para ello, hizo falta una nueva función que permitiera inicializar la matriz de pesos y el vector bias para cada capa oculta de la manera correcta de modo que:

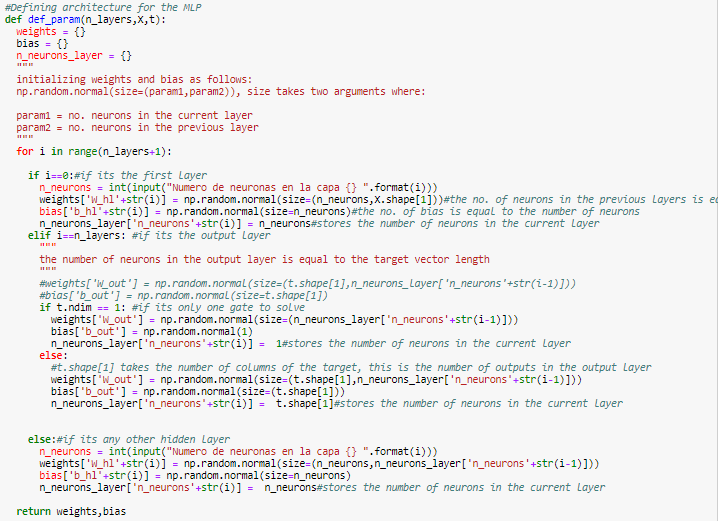


La función que nos permite inicializar la matriz de pesos fue llamada **def\_param(n\_hidden\_layers,x\_train,y\_train)** donde:

**n\_hidden\_layers:** Número de capas ocultas para la red.

**x\_train:** Matriz de vectores de entrenamiento.

**y\_train:** Matriz de etiquetas para cada vector de entrenamiento.



Usando diccionarios para almacenar cada matriz de pesos y bias en una misma variable para la red, preguntamos al usuario para cada capa, el número de neuronas que desee en dicha capa, de modo que, como ya se explico anteriormente en el punto **2.** la matriz de pesos de esa capa se compone de **rows** = número de neuronas deseadas en esa capa y **cols** = número de neuronas en la capa anterior o número de entradas del vector de entrada en caso de ser la primera capa oculta, para el vector de bias, este es del mismo tamaño que el número de neuronas.

Se hizo uso de un diccionario adicional que guarda el número de neuronas de cada capa oculta que se inicializaba, de ese modo, para la capa siguiente era fácil acceder al número de neuronas de la capa anterior.

**Entrenamiento**

Una vez listas las funciones que permiten inicializar la arquitectura de la red neuronal pasamos a las funciones de entrenamiento, divididas en

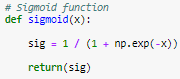
* **Feedforward:** Dado un patrón(vector) de entrada, se realizan las operaciones correspondientes que recorren desde la primer hasta la última capa oculta para obtener un resultado en la capa de salida.
* **Backpropagation:** El error obtenido de la capa de salida es calculado, y se calculan los errores de cada capa oculta, partiendo desde la última capa, hasta la primera.
* **Update Weights and Bias:** Haciendo uso del principio de descenso del gradiente, una vez obtenidos los errores de cada capa, los usamos para actualizar los pesos de forma que nos permita disminuir el error con este cambio.

**Feedforward**

Esta operación está constituida por la propagación de la entrada hacia las capas ocultas, a través de las operaciones de producto punto de matrices y función de activación en cada neurona de cada capa, para ello se usa:

* **np.matmul(matriz1,matriz2):** que devuelve el vector resultante del producto punto de matriz1 por matriz2
* **sigmoid(x):** que dado el vector x, aplica la función sigmoide para saber que neuronas se están activando.

De este modo las funciones quedan:



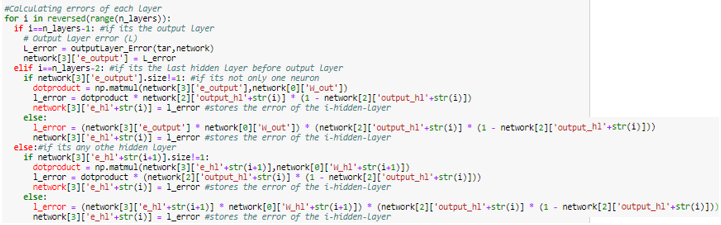


Se ingresa un vector x y este es responsable de comenzar la propagación hacia delante, a través de todas las capas de la red, es importante recalcar que para realizar el producto punto entre la matriz de pesos y la entrada que recibe cada capa, se debe hacer la transpuesta de cada matriz de pesos, pues esto nos permite respetar la regla para realizar el producto punto donde el número de columnas de la matriz1 debe ser igual al número de filas de la matriz2, el resultado es ingresado a la función de activación y este almacenado en un vector de salida que posteriormente será usado para el cálculo de los errores en cada capa.

**En este caso se debe entender que cada neurona está representada por cada fila de la matriz de pesos de su respectiva capa, de modo que cada columna de cada fila corresponde al peso conectado a esa neurona con las neuronas de la capa anterior.**

**Backpropagation**

La función backpropagation nos permite calcular el error que cada capa tuvo hasta la capa de salida con respecto a los pesos que se tienen hasta el momento, derivado de la regla de la cadena, podemos obtener la formula que nos permite dar con el error de cada capa de la red neuronal, en este caso se realiza un bucle que recorra la última capa de la red(capa de salida) hasta la primer capa oculta, pues los errores de cada capa dependen del error de la capa posterior.

La función queda:

**Update weights and bias**

La función hace uso de los errores obtenidos en cada capa, y la salida de la capa que provoco ese error, todo esto obtenido gracias al principio del descenso del gradiente.

**Una parte importante para destacar aquí es que se hizo uso de un concepto llamado “broadcasting” referido a la operación de vectores (sea suma, resta, multiplicación), debemos entender que estas operaciones entre matrices son elemento a elemento, lo cual exige que sea entre matrices del mismo tamaño, sin embargo, broadcasting nos evita el trabajo de siempre tener matrices del mismo tamaño.**

**Broadcasting tiene como concepto que estas operaciones podrán ser realizadas siempre en los siguientes casos:**

* **Caso uno: un vector columna(VC) (n,1) es decir n filas y 1 columna que realiza una operación a un vector fila(VF) (1,m) 1 fila y m columnas, de modo que el vector resultante es una matriz de (n,m) dado que el vector VC se recorrió m veces a través del vector VF para cubrir las m componentes y el vector VF recorrió n veces para cubrir todas las componentes del VC.**

