

Contents

1	Introducción	1
2	Neurona Artificial	3
2.1	Redes Neuronales Artificiales	8
3	Caso de Estudio	10
3.1	Problema	10
3.2	Justificacion	11
3.3	Propuestas de Red Neuronal	12
3.3.1	Topologia	12
3.3.2	Reglas	13
3.3.2.1	Regla de propagación	13
3.3.2.2	Regla de Activacion	14
3.3.2.3	Regla de Salida	14
3.3.2.4	Regla de Aprendizaje	14
3.3.2.4.1	Back Propagation	15
3.3.2.4.2	Metodo de Segundo Orden (Levenger - Marquardt)	16
3.3.2.4.3	R-PROP	18
3.4	Desarrollo de la solucion	19
3.4.1	Herramientas	20
3.4.1.1	Python	20
3.4.1.2	PyCharm	21
3.4.1.3	Package	21
3.4.2	Implementacion	22
3.4.2.1	Back Propagation	22
3.4.2.2	R-PROP	24
3.4.2.3	Segundo Orden	27
3.5	Conclusiones	27

1 Introducción

El cerebro humano es un sistema de cálculo muy complejo, puede llevar a cabo procesamiento que a primera vista parecen sencillos, como por ejemplo,

el reconocimiento de imágenes. Esta capacidad que tiene el cerebro humano para pensar, recordar y resolver problemas ha inspirado a muchos científicos a intentar imitar estos funcionamientos.

Los intentos de crear un ordenador que sea capaz de emular estas capacidades ha dado como resultado la aparición de las llamadas Redes Neuronales Artificiales o Computación Neuronal.

El principal objetivo del Reconocimiento de patrones es la clasificación ya sea supervisada o no supervisada. Aplicaciones como Data Mining, Web Searching, recuperación de datos multimedia, reconocimiento de rostros, reconocimientos de caracteres escritos a mano, etc., requieren de técnicas de reconocimiento de patrones robustas y eficientes. Las redes neuronales, por su capacidad de generalización de la información disponible y su tolerancia al ruido, constituyen una herramienta muy útil en la resolución de este tipo de problemas.[1]

Este documento muestra los conceptos básicos de las Redes Neuronales y su regla de aprendizaje, en particular la configuración en *Perceptrón Multicapa* y el varios algoritmos de aprendizaje (Propagación hacia atrás, Métodos de segundo orden, RPROP, Algoritmos Genéticos).

2 Neurona Artificial

Uno de los retos más importantes a los que se enfrenta el ser humano de nuestra generación es el de la construcción de sistemas inteligentes, en su afán de conseguir este propósito aparecen las redes neuronales artificiales. Desde el punto de vista biológico las RNA son un modelo matemático acerca del funcionamiento del cerebro. *”Los sencillos elementos de cálculo aritmético equivalen a las neuronas-células que procesan la información en el cerebro- y la red en general equivale a un conjunto de neuronas conectadas entre sí”* [2]

Para la raza humana sigue siendo un misterio el funcionamiento del cerebro humano y como se genera el pensamiento, sin embargo años y años de investigación han dado ideas sobre el accionar del mismo. Si se quieren reproducir las acciones del cerebro humano, se debe tener la idea de como funciona. Una explicación sencilla y clara se encuentra en [2]:

”Sabemos que la neurona, o célula nerviosa, es la unidad funcional básica de los tejidos del sistema nervioso, incluido el cerebro. Las neuronas están formadas por el cuerpo de la célula, o soma, en donde se aloja el núcleo de la célula. Del cuerpo de la célula salen ramificaciones de diversas fibras conocidas como dendritas y sale también una fibra más larga denominada axón. Las dendritas se ramifican tejiendo una tupida red alrededor de la célula, mientras el axón se extiende un buen tramo: por lo general, un centímetro (100 veces el diámetro del cuerpo de la célula) y, en casos extremos, hasta un metro. Finalmente, el axón también se ramifica en filamentos y subfilamentos mediante los que establece conexión con las dendritas y los cuerpos de las células de otras neuronas. A la unión o conexión se le conoce como sinapsis. Cada neurona establece sinapsis desde con una docena de otras neuronas hasta con cientos de miles de otras de ellas”

La neurona artificial se ha diseñado como una abstracción de la neurona biológica y se muestra en la Figura 1. La figura representa la neurona j que recibe entradas. Sus partes son:

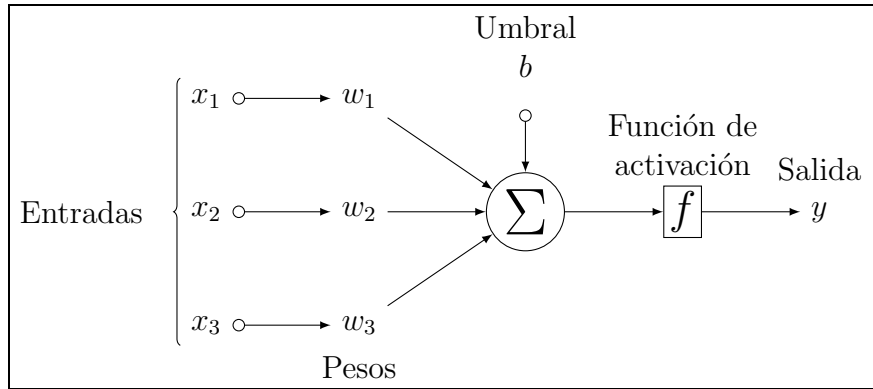


Fig. 1: Representación de una Red Neuronal Artificial.

1. Las **entradas** x_i , que son puntos por los que se reciben los datos provenientes del entorno o bien de otras neuronas. Para este caso se consideran n entradas, siendo el valor de $n = 3$

$$X = (x_1, x_2, x_3)$$

2. La salida y_i . En una neurona biológica corresponde al axón
3. Al igual que en una neurona biológica, la neurona artificial debe permitir establecer conexiones (sinápsis) entre las entradas (dendritas) de una neurona y la salida (axón) de otra. Esta conexión representa con una línea que tiene asociado un valor llamado **peso sináptico** w_{ji} . Nótese que el primer subíndice indica la neurona a la que llega la conexión, mientras que el segundo subíndice indica de donde viene la conexión. El peso representa el factor de importancia de la conexión en la determinación del valor de salida. El valor w_{ji} , que es un número real, se modifica durante el entrenamiento de la red neuronal y es la variable que almacenará la información que indicará que la red ha aprendido algo y por tanto que sirva para un propósito u otro.
4. En la Figura 1 también se observa una entrada especial, llamada umbral, con un valor fijo que puede ser -1 o 1, y con un peso asociado llamado w_0 . El valor del umbral se ajusta igual que cualquier otro peso durante el proceso de entrenamiento.
5. Una **regla de propagación**. Para cierto valor de las entradas x_i y los pesos sinápticos asociados w_{ji} , se realiza algún tipo de operación

para obtener el valor del potencial *post-sináptico* . Este valor es función de las entradas y los pesos. Una de las operaciones mas comunes es realizar la suma ponderada, que no es otra cosa que la sumatoria de las entradas, pero teniendo en cuenta la importancia de cada una (el peso sináptico asociado). Luego:

$$u = \sum_i^{n=3} w_{ji}x_i + w_0b$$

6. Una **función de activación**. Luego de realizar la suma ponderada, se aplica al resultado la función de activación, que se escoge de tal manera que permita obtener la forma deseada para el valor de la salida.

$$\begin{aligned} y &= f(u) \\ &= f\left(\sum_i^{n=3} w_{ji}x_i + w_0b\right) \\ &= f(W.X) \\ &= f(W^T X) \end{aligned} \tag{1}$$

donde las últimas dos ecuaciones están en notación vectorial. Es necesario especificar la función de activación f . Las funciones más usuales se observan en la 10

Con estas especificaciones, se puede ahora explicar cómo funciona la neurona. Se supone en el modelo de neurona más simple, que corresponde a la función de activación escalón, también llamada limitador duro. En este caso, la salida puede tomar solo dos valores -1 y +1 donde la salida está determinada por

$$f(x) = \begin{cases} -1 & \text{if } u < 0 \\ +1 & \text{if } u \geq 0 \end{cases} \tag{2}$$

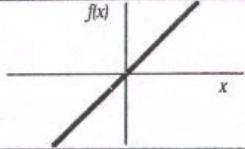
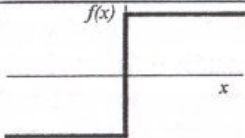
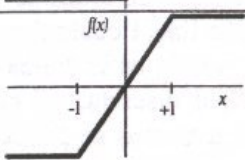
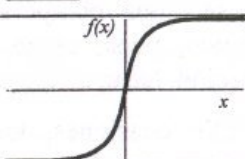
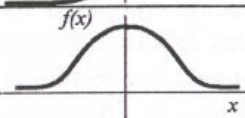
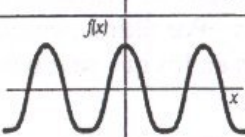
	Función	Rango	Gráfica
Identidad	$y = x$	$[-\infty, +\infty]$	
Escalón	$y = \text{sign}(x)$ $y = H(x)$	$\{-1, +1\}$ $\{0, +1\}$	
Lineal a tramos	$y = \begin{cases} -1, & \text{si } x < -l \\ x, & \text{si } -l \leq x \leq +l \\ +1, & \text{si } x > +l \end{cases}$	$[-1, +1]$	
Sigmoidea	$y = \frac{1}{1+e^{-x}}$ $y = \text{tgh}(x)$	$[0, +1]$ $[-1, +1]$	
Gaussiana	$y = Ae^{-Bx^2}$	$[0, +1]$	
Sinusoidal	$y = A \text{sen}(\omega x + \varphi)$	$[-1, +1]$	

Fig. 2: Funciones de activación.

Entonces, para la función sigmoidea, se tiene que

$$y = \left(\frac{1}{1 + e^{-u}} \right) \quad (3)$$

$$u = \sum_{i=1}^{n=3} w_{ji}x_i + w_0b$$

y para el segundo caso de la función sigmoidea

$$y = \tanh(u) = \left(\frac{e^u - e^{-u}}{e^u + e^{-u}} \right) \quad (4)$$

La expresión de la ecuación que almacena la neurona en virtud del vector de pesos W es el **modelo** que representa en mayor o menor grado el comportamiento del vector de salida y con respecto al vector de entrada X

Entonces, una neurona artificial es un procesador elemental. Se encarga de procesar un vector de n entradas para producir un único valor de salida y . El nivel de activación depende de las entradas recibidas y de los valores sinápticos. Para calcular el estado de activación se ha de calcular en primer lugar la entrada total a la célula. Este valor se calcula como la suma de todas las entradas ponderadas por ciertos valores dados a la entrada.

2.1 Redes Neuronales Artificiales

La capacidad de modelar funciones más complejas aumenta grandemente cuando la neurona no trabaja sola, sino interconectada con otras neuronas, formando Redes Neuronales Artificiales (RNA), tal como se observa de manera simplificada en la Figura 4

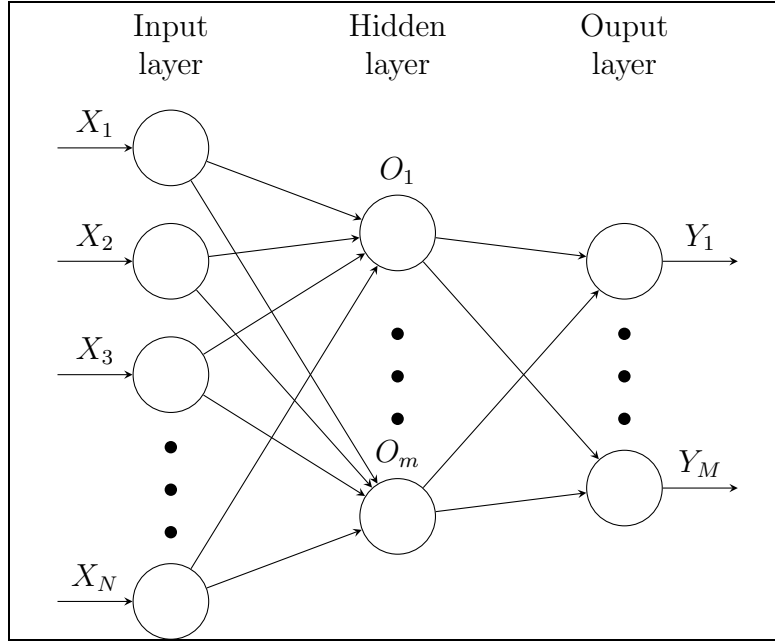


Fig. 3: Topología de la Red Perceptron Multicapa.

La red mas simple se llama **perceptron multicapa**. Esta red define una relación entre las variables de entrada y las variables de salida. Esta relación se obtiene *propagando* hacia adelante los valores de las variables de entrada. Cada neurona procesa la información recibida por sus entradas y produce una respuesta o activación que se propaga, a través de las conexiones correspondientes, hacia las neuronas de la siguiente capa.

Sea un perceptron multicapa con C capas, de las cuales una es la capa de entrada, una la capa de salida y $C - 2$ capas ocultas. Se tiene n_c neuronas en la capa c , con $c = 1, 2, 3, \dots, C$. sea $W^c = (w_{ji}^c)$ la matriz de pesos asociada a las conexiones de la capa c a la capa $c+1$ para $c = 1, 2, 3, \dots, C-1$, donde w_{ji}^c representa el peso de la conexión de la neurona i de la capa C a la neurona

j de la capa $C + 1$. Sea $U^c = (u_j^c)$ el vector de umbrales de las neuronas de la capa c para $c = 2, 3, \dots, C$. Se denota o_j^c a la activación de la neurona j de la capa c ; estas activaciones se calculan del siguiente modo:

1. Entrada (o_i^1). Estas neuronas transmiten hacia la red las señales recibidas del exterior

$$o_j^1 = x_j \quad \text{para} \quad j = 1, 2, 3, \dots, n_1 \quad (5)$$

Donde $X = (x_1, x_2, \dots, x_n)$ representan el vector de entrada a la red.

2. Activación de las neuronas de la capa oculta $c(o_j^c)$: Las neuronas ocultas procesan la información recibida aplicando la función de activación f a la suma de los productos de las activaciones que recibe por sus correspondientes pesos:

$$o_j^c = f\left(\sum_{i=1}^{n_{c-1}} w_{ji}^{c-1} o_i^{c-1} + u_j^c\right) \quad (6)$$

Para $j = 1, 2, 3, \dots, n_c$ y $c = 2, 3, \dots, c - 1$

3. Salida (a_i^C) : Al igual que en las capas ocultas, la activación de estas neuronas viene dada por la función de activación f

$$y_j = o_j^c = f\left(\sum_{i=1}^{n_{c-1}} w_{ji}^{c-1} o_i^{c-1} + u_j^c\right) \quad (7)$$

Para $j = 1, 2, 3, \dots, n_c$ donde $Y = (y_1, y_2, y_3, \dots, y_c)$ es el vector de salida de la red.

Para el perceptrón multicapa las funciones de activación más usadas son la funciones **sigmoidal**:

$$f(u) = \frac{1}{1 + e^{-u}} \quad (8)$$

y la función **tangente hiperbólica**

$$f(u) = \left(\frac{1 - e^{-u}}{1 + e^{-u}}\right) \quad (9)$$

Estas funciones tienen una forma similar pero se diferencian en que la sigmoideal tiene un rango continuo de valores dentro de los intervalos $[0, 1]$ mientras que la tangente hiperbólica tiene un rango continuo en el intervalo $[-1, 1]$.

3 Caso de Estudio

Todos los días, millones de e-mails invaden las bandejas de entrada de los usuarios de Internet. De todos éstos, una cantidad muy importante es considerada "correo basura". Compuesto por mensajes publicitarios no solicitados, cadenas de la suerte o incluso virus que se autoenvían, el spam afecta a más de un usuario, y hace que la tarea de revisar el correo sea una verdadera molestia. El problema fundamental lo representan los spams, que son mensajes publicitarios no solicitados. Ya no resulta raro para quienes contamos con una dirección de correo electrónico recibir a diario varios mensajes con propagandas de las más variadas temáticas. A pesar de que ningún método de detección de Spam es totalmente efectivo, consideramos que si es posible mejorar los existentes y reducir considerablemente las deficiencias que actualmente presentan las herramientas disponibles. Es un hecho que parte de los mensajes no deseados escapan a los sistemas de detección de correo basura constituyendo así un "falso negativo", igualmente existe la posibilidad de identificar un mensaje como Spam sin serlo, lo que se conoce como "falso positivo". La idea es tomar las máximas precauciones posibles para minimizar este efecto, y para ello se debe ser consciente de este hecho antes de adoptar las posibles medidas de filtrado que se propondrá.[3]

3.1 Problema

El crecimiento de Internet a nivel mundial está cambiando nuestra forma de comunicación entre otros, por lo que cada vez la gente utiliza más el correo electrónico. A causa de un número tentativo de correos electrónicos los publicistas y spammers se ven los modos para obtener un listado grande de correos y así poder enviar spam. Todos los días, billones de e-mails invaden las bandejas de entrada de los usuarios de Internet. De todos éstos, una cantidad muy importante es considerada "correo basura". Compuesto por mensajes publicitarios no solicitados, cadenas de la suerte o incluso virus que se auto envían, el spam aqueja a más de un usuario, y hace que la tarea de

revisar correo sea una verdadera molestia.[3]

Los principales problemas son los siguientes:

- Pérdida de productividad y dinero en las empresas
- Amenaza la viabilidad de Internet como un medio efectivo de comunicación.
- Incremento de costos relacionados con el tiempo.
- Genera importantes costos de seguridad a empresas ISP's.
- Incremento de propagación de virus informáticos.
- Saturación de servidores. Muchos servidores dedicados para uso privado o para uso general son congestionados implicando una reducción de calidad de servicio.
- Denegación de servicios (Deny of services). Una cantidad excesiva de correos no deseados pueden congestionar totalmente el servicio y así denegarlo al mismo.
- Buzón de entrada incontrolable por parte del receptor. Causado por la cantidad masiva que los spammers envían a los correos electrónicos.
- Daño de imagen a terceros.
- Molestias por parte del receptor.

3.2 Justificación

El correo electrónico, es sin duda un medio que nos permite comunicar rápidamente ofreciéndonos reducción de tiempo y costo Sin embargo muchas personas aprovechan esto para utilizarlo de forma no legítima con fines publicitarios, ocasionando una serie de problemas a nivel personal como empresarial. Como contramedida a esta acción se necesitan herramientas capaces de reducir el spam. De esta manera es muy importante la elaboración de anti-spams, ya que es la forma más viable de acabar con el spam y ofrecer a los usuarios seguridad y tranquilidad en los correos electrónicos, y por otra parte reducir los costos para las empresas ISP's y controlar la saturación de

servidores de correo electrónico. **El desarrollo de una herramienta informática capaz de aminorar con los problemas que causa el spam**, no es solamente capaz de **ahorrar mucho dinero** en aquellas empresas que suelen estar perjudicadas con el spam, sino también es capaz de permitir una mejor utilización y minimizar los dolores de cabeza a cualquier usuario del correo electrónico.[3]

3.3 Propuestas de Red Neuronal

Las redes neuronales tienen un gran poder en los algoritmos de Machine Learning. Ellos pueden ser usados para transformar las características como una forma bastante compleja de decisión no lineal. Dentro de estos se encuentran el perceptrón multicapa que en primera instancia se puede utilizar para problemas de clasificación.

Como el problema que estamos tratando en este documento es un problema común de clasificación donde la red neuronal deberá clasificar si un determinado email es o no es spam. En este problema, nosotros estamos dando un paquete de correos (que ya están normalizados.) and la red neuronal decidirá si cada email es o no es spam. Después de determinar los pesos fijos de la red le presentaremos un nuevo conjunto de correos (test data) y evaluaremos si estos correos son o no correos spam.

3.3.1 Topología

Tenemos tres capas en la red neuronal para la detección de spam.

- La primera capa son los datos de entrada que tienen 57 neuronas, 1 neurona por cada característica del correo.
- La segunda capa es la capa intermedia que tiene 4 neuronas
- La capa final o de salida tiene 1 sola neurona que indica si es o no es spam

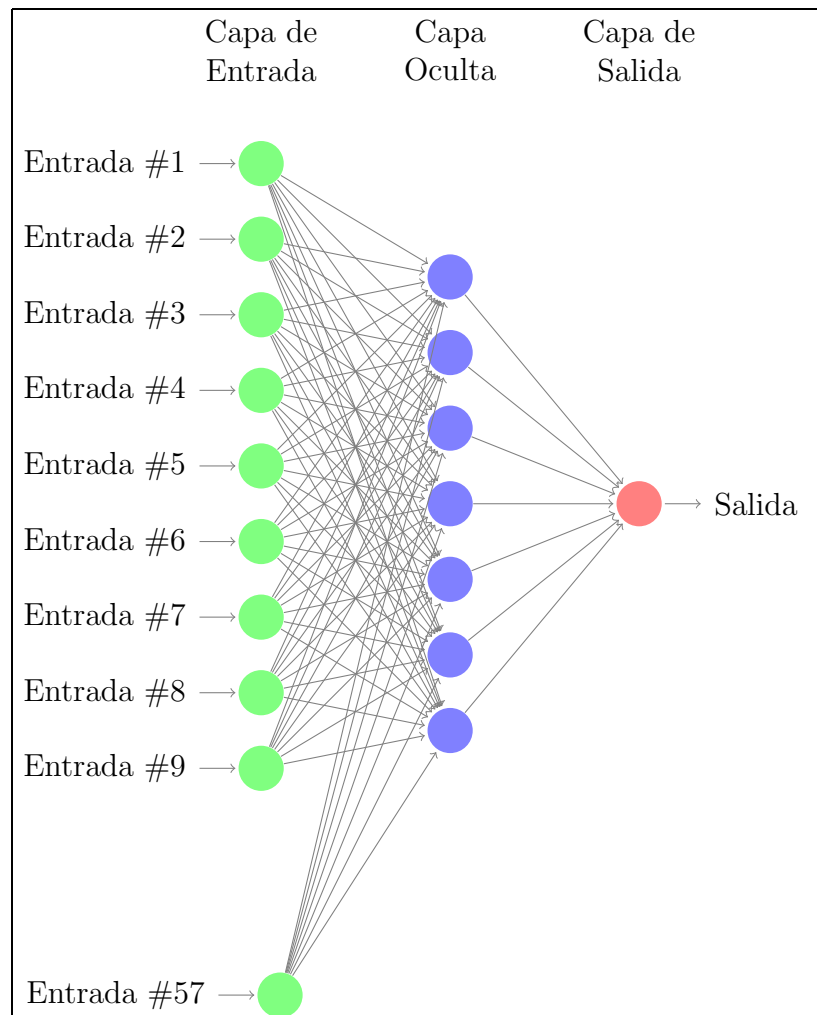


Fig. 4: Topologia de la Red Perceptron Multicapa para la detección de correos spam.

3.3.2 Reglas

3.3.2.1 Regla de propagación

La regla de propagación esta determinada por la sumatoria de las entradas y los pesos, tomando como otra entrada a la neurona el *bias*.

$$u_j^c = \sum_{i=0}^{n_{c-1}} w_{ji}^{c-1} x_i^{c-1} \quad (10)$$

3.3.2.2 Regla de Activacion

La funcion de activación esta determinada por la funcion sigmoide.

$$y = \left(\frac{1}{1 + e^{-u_j^c}} \right)$$

$$u = \sum_{i=0}^n w_{ji} x_i \quad (11)$$

Entonces la salida con esta función, tiene la siguiente forma

$$y = f(u)$$

$$= f\left(\sum_{i=0}^n w_{ji} x_i\right)$$

$$= \left(\frac{1}{1 + e^{-\sum_{i=0}^n w_{ji} x_i}} \right) \quad (12)$$

3.3.2.3 Regla de Salida

Finalmente la regla de salida esta determinada por

$$y_j^c = f(u_j^c)$$

$$= f\left(\sum_{i=0}^{n_{c-1}} w_{ji}^{c-1} x_i^{c-1}\right)$$

$$= \frac{1}{1 + e^{-\sum_{i=0}^{n_{c-1}} w_{ji}^{c-1} x_i^{c-1}}} \quad (13)$$

3.3.2.4 Regla de Aprendizaje

Se han planteado tres reglas de aprendizaje, cada una de estas seran descritas a continuación.

3.3.2.4.1 Back Propagation

Es la regla mas comun, para poder explicar la regla de aprendizaje mediante propagación hacia atras, se tomara como ejemplo la la arquitectura planteada en4

Considerando que tenemos dos capas, dado que la capa de entrada no hace ninguna operación, entonces:

- Capa I

$$\begin{aligned} v_j^I &= \langle w_j^I, X \rangle = \sum_{i=0}^N w_{ji}^I x_i \\ y_j^I &= f(v_j^I) = \frac{1}{1 + e^{-v_j^I}} \end{aligned} \quad (14)$$

- Capa II

$$v_j^I I = \langle w_j^I I, y^I \rangle y_j^I I = f(v_j^I I) \quad (15)$$

Ahora para cada salida que tengamos de la capa II, calculamos el error total cometido, de la siguiente manera:

$$\xi(n) = \frac{1}{2} \sum_{j=1} M e_j^2(n) \quad (16)$$

Elevar al cuadro tiene mucha importancia dado que, imaginemos que en una salida y_x el error sea -1 y en otra salida y_k el error sea -1 al sumarlo el error total resultaría cero, entonces, al elevarlo al cuadrado tiene agunas ventajas como la mencionada.

La regla de progapación hacia atras consiste en modificar los pesos para lograr el menor error posible, para lograr esto debemos recordar la utilidad que tiene el gradiente de una función:

La gradiente o también conocido como vector gradiente, es un campo vectorial. El vector gradiente de una funcion f evaluado en un punto generico x del dominino de f , indica la dirección en la cual el campo f varía mas rápidamente y su módulo representa el ritmo de variación de f en la dirección de dicho vector gradiente.[4]

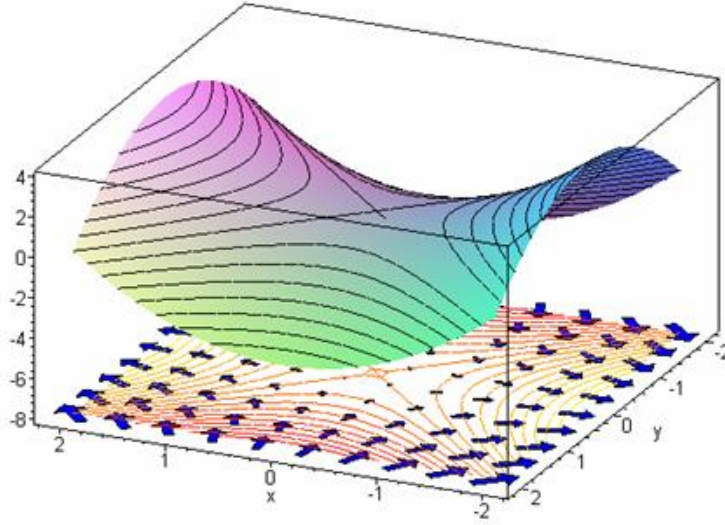


Fig. 5: Gradiente de una función f .

Entonces para lograr disminuir el error, aplicaremos la siguiente variación:

$$\nabla w_{ji}(n) = -\mu \frac{\partial \xi(n)}{\partial w_{ji}(n)} \quad (17)$$

3.3.2.4.2 Metodo de Segundo Orden (Levenger - Marquardt)

El método de Levenger - Marquardt es un tipo de algoritmo de entrenamiento similar a los denominados quasi-Newton. Para entender la funcionalidad del método LM es conveniente empezar por comprender el método de Newton. El método de Newton es un método de segundo orden que constituye una alternativa a los métodos de gradiente conjugado para optimización rápida. El paso básico del método de Newton durante la época n es

$$w(n+1) = w(n) - H(n)^{-1}g(n) \quad (18)$$

donde w es el vector de pesos, $g(n)$ es el vector gradiente actual y $H(n)$ es la matriz Hessiana (de derivadas segundas) de la función de error respecto a los pesos, evaluada en los valores actuales de pesos y umbrales. El método de Newton generalmente converge más rápido que los métodos de gradiente

conjugado. Desafortunadamente, la matriz Hessiana es compleja y costosa de calcular para las redes feedforward. Los algoritmos denominados quasi-Newton (método de la secante) calculan y actualizan una aproximación de la matriz Hessiana sin necesidad de resolver las derivadas de segundo orden en cada iteración. La actualización realizada es función del gradiente. Al igual que los métodos quasi-Newton, el algoritmo LM fue diseñado para acometer entrenamientos rápidos de segundo orden sin tener que calcular la matriz Hessiana. Cuando la función de error tiene la forma de una suma de cuadrados (el caso típico en las redes feedforward), entonces la matriz Hessiana puede aproximarse como

$$H = J^T J \quad (19)$$

donde J es la matriz Jacobiana que contiene las primeras derivadas de la función de error de la red respecto a los pesos y umbrales. El gradiente se obtiene como

$$g = J^T e \quad (20)$$

siendo e el vector de errores de la red. El cálculo de la matriz Jacobiana se reduce a un simple cálculo del algoritmo de retropropagación que es mucho más sencillo que construir la matriz Hessiana. El método LM actualiza los pesos de forma similar al método de Newton pero introduciendo los cambios anteriores:

$$w(n+1) = w(n)[J^T J + \mu I]^{-1} J^T e \quad (21)$$

donde el parámetro μ actúa como tasa de aprendizaje. Cuando μ es cero, el algoritmo se convierte en el método de Newton empleando la forma aproximada del cálculo de la matriz Hessiana:

$$w(n+1) = w(n)H^{-1}g \quad (22)$$

Cuando μ es grande, se puede despreciar la aportación de la matriz Hessiana al cálculo y obtenemos el método de descenso de gradiente con un tamaño de paso $(\frac{1}{\mu})$:

$$w(n+1) = w(n) - \frac{1}{\mu} g \quad (23)$$

El método de Newton es más eficaz cerca de un mínimo de error, así que el objetivo es migrar al método de Newton tan pronto como sea posible.

Así pues, se reduce μ tras cada paso exitoso (cada vez que el error se logra reducir) y solo se aumenta cuando el error aumenta respecto a la iteración anterior. Con esta metodología, la función de error siempre disminuye tras cada iteración del algoritmo. El algoritmo LM parece ser el método más rápido para entrenar redes feedforward de tamaño moderado (hasta varios centenares de parámetros). Su principal desventaja es que requiere almacenar las matrices Jacobianas que, para ciertos conjuntos de datos, pueden ser muy grandes, lo que significa un gran uso de memoria.

3.3.2.4.3 R-PROP

El algoritmo RPROP calcula el cambio de los pesos en forma separada, es guiado por laprimera derivada de f , en este caso; f es una medida de la diferencia entre la salida arrojada por la red neuronal y el valor esperado. RPROP utiliza parámetros independientes que controlan la velocidad con que se recorre la función objetivo para cada uno de los pesos de la red neuronal, al no verse afectado por la saturación de la red neuronal converge mas rápidamente que otros algoritmos

Un perceptron multicapa busca una función y_t que se construye en funcion de sus valores pasados, $y_{t-1}, y_{t-2}, \dots, y_{t-P}$.

$$y_t = \beta_* + \sum_H^{h=1} \beta_h g\left[\frac{1}{2\sigma_y^{-1}}(\alpha_{x,j} + \sum_{p=1}^P \alpha_{p,h} y_{t-p})\right] + \xi_t \quad (24)$$

Donde los parametros $\omega = [\beta_*, \beta_h, \alpha_{*,h}, \alpha_{p,h}]$, $h = 1 \dots H, p = 1 \dots P$ son estimados usando el principio de máxima verosimilitud de los residuales, el cual equivale a la minimización de una función de costo que es definida usualmente como error cuadratico medio. Esta ecuación equivale a un modelo estadístico no paramétrico de regresión no lineal, ξ_t sigue una distribución normal con media cero y varianza desconocida σ^2 ; H representa el número de neuronas en la capa oculta, P es el número de rezagos de la variable dependiente y g es la función de activación de las neuronas de la capa oculta.

El algoritmo RPROP busca encontrar los valores del vector de parámetros ω , de forma que se minimice la diferencia entre los valores reales y_t^* y los valores de y_t .

La actualización de los pesos, viene dada por

$$\Delta w_{ji}(n) = -\alpha(n) \text{signo}(\nabla_{w_{j,i}(n)} E) \quad (25)$$

La utilizacion del signo del gradiente en la actualización de los pesos supone un ahorro en la carga computacional. Por otro lado, la constante de adaptación viene dada por

$$\alpha(n) = \begin{cases} \alpha(n) = \min(\alpha(n)u, \alpha_{max}) \longleftrightarrow (\nabla_{w_{j,i}(n)}E)(\nabla_{w_{j,i}(n-1)}E) > 0 \\ \max(\alpha(n)d, \alpha_{min}) \longleftrightarrow (\nabla_{w_{j,i}(n)}E)(\nabla_{w_{j,i}(n-1)}E) < 0 \end{cases} \quad (26)$$

con $u > 1$ y $d < 1$.

Se intenta tener un valor bajo en las proximidades del mínimo y mayor lejos de éste. controlando de esta manera la velocidad de convergencia, pero se limitan los valores ya que si son demasiado grandes se pueden tener inestabilidades y si son demasiado pequeños la velocidad de convergencia puede ser muy lenta.

3.4 Desarrollo de la solución

Para el desarrollo del perceptrón multicapa planteado anteriormente hemos escogido como primera herramienta R(Figura 7) y con el IDE Rstudio(Figura 6), pero no se utilizó porque **el tiempo de aprendizaje para un número mayor de 10 épocas era muy elevado**, a comparación de la segunda herramienta, python.



Fig. 6: IDE Rstudio



Fig. 7: Lenguaje R

3.4.1 Herramientas

Como se dijo anteriormente no se utilizó la herramienta R debido a su alto tiempo en procesar la información, a continuación se describen las herramientas utilizadas.

3.4.1.1 Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License,¹ que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriores.[5]



Fig. 8: Lenguaje Python.

3.4.1.2 PyCharm

El entorno de desarrollo integrado que se utilizo es PyCharm, es un entorno muy popular utilizado en la comunidad de python que provee una manera sencilla de programar en python eligiendo la version de python que deseamos compilar y agregar los paquetes necesarios para el desarrollo de nuestros proyectos.



Fig. 9: IDE Pycharm.

3.4.1.3 Package

Para el desarrollo de los algoritmos de aprendizaje para el perceptron multi-capas se utilizo una libreria para Inteligencia Artificial llamada NeuPy. Esta libreria actualmente en su version 0.6 soporta diferentes tipos de Redes Neuronales desde un perceptron simple hasta modelos de deep learning [6].

Actualmente soporta las siguientes características:

- Deep Learning
- Reinforcement Learning (RL)
- Convolutional Neuronal Networks (CNN)
- Recurrent Neuronal Networks (RNN)
- Restricted Boltzmann Machine (RBM)
- Multilayer Perceptron (MLP)
- Networks based on the Radial Basis Functions (RBFN)
- Ensemble Networks

- Competitive Networks
- Basic Linear Networks
- Regularization Algorithms
- Step Update Algorithms

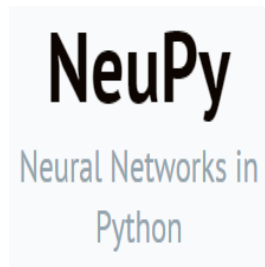


Fig. 10: Funciones de activación.

3.4.2 Implementacion

A continuación pasaremos a describir los métodos de aprendizaje para nuestro perceptron multicapa.

3.4.2.1 Back Propagation

Para describir mejor el procedimientos pasaremos de describir el codigo por partes:

1. Se procede a importar los paquetes necesarios para la implementación, creamos las funciones de activación y la funcion derivada

```
from sklearn import preprocessing
import numpy as np

def derivative(x):
    return x * (1.0 - x)

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

x = []
y = []
```

Fig. 11: Importar paquetes necesarios.

- Ahora leeremos nuestro archivo *Train.csv* que contiene los datos normalizados

```
# read the training data
with open('Train.csv') as f:
    for line in f:
        curr = line.split(',')
        new_curr = [1]
        for item in curr[:len(curr) - 1]:
            new_curr.append(float(item))
        X.append(new_curr)
        Y.append([float(curr[-1])])
```

Fig. 12: Leer la data de entrenamiento.

- Asignamos los valores a nuestros arreglos para poder guardar los valores locales que se van creando, al igual que los valores random para los pesos iniciales

```
X = np.array(X)
X = preprocessing.scale(X) # feature scaling
Y = np.array(Y)
# the first 2500 out of 3000 emails will serve as training data
X_train = X[0:4025]
Y_train = Y[0:4025]
# the rest 500 emails will serve as testing data
X_test = X[4025:]
Y_test = Y[4025:]
X = X_train
Y = Y_train
# we have 3 layers: input layer, hidden layer and output layer
# input layer has 57 nodes (1 for each feature)
# hidden layer has 4 nodes
# output layer has 1 node
dim1 = len(X_train[0])
dim2 = 4
```

Fig. 13: Asignar valores de entrada a nuestros arreglos.

- Llamamos al metodo de Rprop de la libreria neuPy para poder pasarle los parametros de entrada y las configuraciones necesarias

```
np.random.seed(1)
weight0 = 2 * np.random.random((dim1, dim2)) - 1
weight1 = 2 * np.random.random((dim2, 1)) - 1
# you can change the number of iterations
for j in range(25000):
    # first evaluate the output for each training email
    layer_0 = X_train
    layer_1 = sigmoid(np.dot(layer_0, weight0))
    layer_2 = sigmoid(np.dot(layer_1, weight1))
    # calculate the error
    layer_2_error = Y_train - layer_2
    # perform back propagation
    layer_2_delta = layer_2_error * derivative(layer_2)
    layer_1_error = layer_2_delta.dot(weight1.T)
    layer_1_delta = layer_1_error * derivative(layer_1)
    # update the weight vectors
    weight1 += layer_1.T.dot(layer_2_delta)
    weight0 += layer_0.T.dot(layer_1_delta)
```

Fig. 14: Crear las epocas y comenzar a iterar.

5. Finalmente Podemos mostrar los valores que se han obtenido luego del entrenamiento y del recuerdo

```
# evaluation on the testing data
layer_0 = X_test
layer_1 = sigmoid(np.dot(layer_0, weight0))
layer_2 = sigmoid(np.dot(layer_1, weight1))
correct = 0

# if the output is > 0.5, then label as spam else no spam
for i in range(len(layer_2)):
    if(layer_2[i][0] > 0.5):
        layer_2[i][0] = 1
    else:
        layer_2[i][0] = 0
    if(layer_2[i][0] == y_test[i][0]):
        correct += 1

# printing the output
print("total = ", len(layer_2))
print("correct = ", correct)
print("accuracy = ", correct * 100.0 / len(layer_2))
```

Fig. 15: Imprimir los resultados.

3.4.2.2 R-PROP

Para describir mejor el procedimientos pasaremos de describir el código por partes:

1. Se procede a importar los paquetes necesarios para la implementación

```
from sklearn import preprocessing, metrics
import numpy as np
from neupy import algorithms, layers, plots
```

Fig. 16: Importar paquetes necesarios.

2. Ahora asignaremos nuestros vectores de entrada y salida respectivamente.

```
X = []
Y = []
```

Fig. 17: Inicializar vectores.

3. Se procede a leer la data de entrenamiento

```
# read the training data
with open('Train.csv') as f:
    for line in f:
        curr = line.split(',')
        new_curr = []
        for item in curr[:len(curr) - 1]:
            new_curr.append(float(item))
        X.append(new_curr)
        Y.append([float(curr[-1])])
```

Fig. 18: Leer data de entrenamiento.

4. Asignamos los valores a nuestros arreglos para poder guardar los valores locales que se van creando

```
X = np.array(X)
X = preprocessing.scale(X) # feature scaling
Y = np.array(Y)

# los primero 3067 de los 4601 datos serviran para el entrenamiento
x_train = X[0:3067]
y_train = Y[0:3067]

# el resto de los datos serviran para la validacion (1534)
x_test = X[3067:]
y_test = Y[3067:]
```

Fig. 19: Asignar valores de entrada a nuestros arreglos.

5. Llamamos al metodo de Rprop de la libreria neuPy para poder pasarle los parametros de entrada y las configuraciones necesarias

```
#se crea la red neuronal con la arquitectura 57-7-1
rpropnet = algorithms.RPROP(
    [
        layers.Input(57),
        layers.Sigmoid(7),
        layers.Sigmoid(1),
    ],
    error='mse',
    verbose=True,
    shuffle_data=True,
    maxstep=1,
    minstep=1e-7,
)
```

Fig. 20: Llamar al constructor del RPROP.

6. Finalmente Podemos mostrar los valores que se han obtenido luego del entrenamiento y del recuerdo

```
#se realiza el entrenamiento de la red
rpropnet.train(input_train=x_train,target_train=y_train,epochs=200)

#se muestra un grafico de los errores cometidos en el entrenamiento
plots.error_plot(rpropnet)

y_train_predicted = rpropnet.predict(x_train).round()
y_test_predicted = rpropnet.predict(x_test).round()

# se muestran las predicciones
print(metrics.classification_report(y_train_predicted, y_train))
print(metrics.confusion_matrix(y_train_predicted, y_train))
print()
print(metrics.classification_report(y_test_predicted, y_test))
print(metrics.confusion_matrix(y_test_predicted, y_test))
```

Fig. 21: Imprimir las salidas.

3.4.2.3 Segundo Orden

3.5 Conclusiones

References

- [1] Laura Lanzarini. Redes neuronales aplicadas al reconocimiento de patrones. http://sedici.unlp.edu.ar/bitstream/handle/10915/22061/Documento_completo.pdf?sequence=1, 2017. [Online; accessed 15-October-2017].
- [2] P Russell, S.J.; Norvig. *Inteligencia Artificial*. Prentice Hall Hispanoamerica, 1996.
- [3] Hugo Galán Asensio. Inteligencia artificial.redes neuronales y aplicaciones. *Estudiantes*, 2010.
- [4] Wikipedia. Gradiente. <https://es.wikipedia.org/wiki/Gradiente>, 2017. [Online; accessed 16-October-2017].
- [5] Python Software Foundation. The official home of the python programming language. <https://www.python.org/>, 2017. [Online; accessed 16-October-2017].
- [6] Yurii Shevchuk. Python library for artificial neural networks. <http://neupy.com/pages/home.html>, 2017. [Online; accessed 16-October-2017].