DESARROLLO DE LIBRERÍA PARA MANEJO DE REDES NEURONALES EN JAVA PARA TECNOFACTOR

Development of a library for managing Neural Networks in Java for TECNOFACTOR

JUAN CAMILO ARBOLEDA ECHEVERRY

Proyecto de Grado

Asesor

Rafael David Rincón

UNIVERSIDAD EAFIT

MEDELLÍN

ESCUELA DE INGENIERÍAS

INGENIERÍA DE SISTEMAS

2018

CONTENIDO

LISTA DE FIGURAS

RESUMEN

En el presente proyecto de grado se pretende diseñar y desarrollar una librería de trabajo que pueda ser utilizada para la creación y entrenamiento de Redes Neuronales, definida de manera que su uso sea sencillo por parte de desarrolladores Java.

Se utilizará la librería desarrollada para diseñar y obtener una red neuronal capaz de reconocer dígitos escritos a mano, a partir de la base de datos MNIST.

Palabras clave: Redes neuronales, Machine Learning, Inteligencia artificial, reconocimiento de patrones, librería, lenguaje Java.

ABSTRACT

The aim of the present project is to design and develop a working library that enables the creation and adaptation of Neural Networks, defined in a way that is simple to use by Java developers.

The developed library will be used to design and obtain a neural network capable of recognizing handwritten digits, from the MNIST database.

Keywords: Neural networks, Machine Learning, Artificial Intelligence, Pattern recognition, Java.

INTRODUCCIÓN

A medida que las computadoras evolucionan, en su software al igual que en su hardware, se hace más evidente la similitud que podrían llegar a tener con el cerebro humano, tanto en su estructura como en la forma como procesan la información, puesto que, con los avances realizados en el campo de la Inteligencia Artificial, se han desarrollado diversas metodologías en el aprendizaje de las máquinas para toma de decisiones. Una de ellas es el uso de Redes Neuronales, "las cuales pueden realizar cálculos que se asemejan a características biológicas del cerebro" (Diaz, Ghaziri, & Glover, 1996) para resolver problemas que han sido de gran dificultad con técnicas clásicas de computación, por ejemplo, el reconocimiento de patrones o imágenes, la interpretación de lenguaje hablado, "o complejos problemas de tipo combinatorio" (Diaz, Ghaziri, & Glover, 1996).

Uno de los tópicos que ha tomado mayor relevancia en los últimos años en el mundo de la informática es el de la Inteligencia Artificial, siendo el desarrollo de Redes Neuronales una de las más importantes tecnologías para el aprendizaje de las máquinas. El desarrollo y la investigación en Redes Neuronales ha permitido abordar problemas que con las metodologías clásicas de la Ingeniería de Sistemas ha sido casi imposibles de resolver.

Uno de estos problemas es el reconocimiento de patrones visuales, como el de texto en imágenes o caracteres escritos a mano, cuya solución a través del desarrollo y entrenamiento de Redes Neuronales ha venido siendo optimizada con los años. Hoy en día se pueden encontrar soluciones comerciales para el reconocimiento de documentos, tarjetas de presentación, identificación de rostros como lo hace Facebook, e incluso reconocimiento de voz e

interpretación del lenguaje hablado en asistentes personales, como Siri de Apple o Alexa de Amazon.

Una de las características que se encuentra en la frontera del conocimiento en Redes Neuronales es el de simular, con una mayor precisión, el comportamiento y la estructura biológica del cerebro humano. La primera neurona artificial fue definida en 1943 por Warren McCulloh y Walter Pitts (Nielsen, s.f.). Desde ese entonces, se han realizado importantes investigaciones y desarrollos, principalmente en la definición de las arquitecturas de las redes neuronales y en la forma de optimizar su aprendizaje.

Este tema cobra importancia para la ingeniería de Sistemas, pues es una frontera del conocimiento con demasiadas potenciales aplicaciones, al igual que para la biología, la medicina y la neurología, pues es una forma no invasiva en que se puede comprender mejor el funcionamiento del cerebro.

El presente trabajo puede servir como base para futuras investigaciones y/o implementaciones, pues dentro de sus objetivos se encuentra el de definir y desarrollar una librería en Java, lo suficientemente genérica, para ser utilizada en múltiples situaciones que involucren Machine Learning. Al igual que la definición de una red neuronal para identificar dígitos escritos a mano, puede incluirse, en conjunto con otras funcionalidades, en una aplicación más robusta y con un alcance mayor.

PLANTEAMIENTO DEL PROBLEMA

TECNOFACTOR es una compañía colombiana fundada en el año 2006, especializada en el diseño y desarrollo de aplicaciones en entornos Web para grandes empresas. Se ha dedicado principalmente al desarrollo de software a la medida, en proyectos para clientes en Colombia y latino-américa, como Solunion, Mapfre, Domina Entrega Total, el hospital Pablo Tobón Uribe, entre otros.

Aunque el desarrollo de software a la medida es la principal fortaleza de TECNOFACTOR, dentro de los objetivos a mediano plazo de la compañía, se encuentra el de desarrollar un portafolio de productos de marca propia para ampliar su oferta y aumentar su reconocimiento en el mercado. Quiere construir productos ***innovadores*** que implementen nuevas tendencias tecnológicas, y que en realidad marquen una diferencia con otras opciones existentes.

Para esto, ha definido que uno de los campos en los que quiere incursionar es el de Machine Learning, siendo uno de sus primeros proyectos la implementación de interpretadores de textos y reconocimiento de patrones, pues su necesidad inicial es la creación de un identificador de documentos tipo factura, para reconocerlos de manera automática y extraer la información básica necesaria para otros procesos, como facturación electrónica, factoring y confirming, sin importar el sistema origen de la información.

Ya que el deseo de TECNOFACTOR es comercializar sus productos en el sector privado, se requiere la implementación de un Framework en Java con el cual no se tengan conflictos por licenciamiento, además que pretende realizar uno propio, porque las opciones disponibles encontradas, como TensorFlow (TensorFlow, s.f.), Caffe (Jia, s.f.) o Theano (LISA lab., 2008-

2017), desarrolladas en Python; o Neuroph (SourceForge, s.f.), Encog (Heaton Research, Inc, 2018) o Deeplearning4j (Skymind, 2018), desarrolladas en Java, requerirían ser adaptadas en su arquitectura para su utilización en las soluciones actuales de Tecnofactor, además que al estar implementadas en modelos matemáticos matriciales, cuyos cálculos son realizados a partir de operaciones con matrices, deberían cambiarse casi en su totalidad, para poder realizar cálculos en neuronas individuales o grupos de neuronas según la necesidad, y no masivamente en una multiplicación matricial, dificultando así el escalamiento del modelo de Machine Learning de la manera en que lo tiene previsto la empresa.

En el presente trabajo de grado se pretende desarrollar una librería escrita en Java, para la creación y utilización de redes neuronales que pueda ser utilizada con fines comerciales, y que permita futuras adaptaciones y escalamientos en el campo de Machine Learning.

Se incluirá en el trabajo de grado, la implementación de una red neuronal para reconocer dígitos escritos a mano, utilizando la base de datos de dígitos escritos a mano MNIST para entrenarla, dado que este problema se ajusta muy bien como paso inicial a la solución requerida por TECNOFACTOR; al igual que será la base para demostrar y medir el funcionamiento de la librería desarrollada.

JUSTIFICACIÓN

Las redes neuronales pueden ser utilizadas para dar solución a diversos tipos de problemas, que con técnicas clásicas de computación podría ser muy difícil su implementación, como, por ejemplo, el reconocimiento de patrones e imágenes o la interpretación de lenguaje hablado. Las soluciones se diferencian en el tipo de entrenamiento que se realice a la Red Neuronal y los datos que se utilicen para hacerlo (Diaz, Ghaziri, & Glover, 1996).

En el desarrollo de las aplicaciones empresariales que se encuentran típicamente en el mercado colombiano, actualmente no existen muchas que en realidad aprovechen las ventajas que pueden ofrecer las diferentes técnicas de IA y Machine Learning, además que, dado el incremento constante en la competitividad de las empresas del medio, se presenta una necesidad propia de innovación y de mejora en las soluciones que se construyen. Cada día, la IA tiene una mayor presencia en las aplicaciones y dispositivos que utilizamos.

TECNOFACTOR ha reconocido la necesidad de incursionar en el mundo de la Inteligencia Artificial y ampliar sus bases metodológicas y técnicas de desarrollo, ya que esto le representará una ventaja competitiva frente al medio, por lo que, en el presente proyecto de grado, se pretende diseñar y desarrollar un framework de trabajo que facilite la comprensión, creación y adaptación de Redes Neuronales, en un entorno fácil de utilizar para desarrolladores Java.

Particularmente para TECNOFACTOR, representa una herramienta que sus desarrolladores pueden utilizar de una manera natural en su entorno actual de desarrollo, para ampliar el abanico de soluciones que ofrece.

Este tipo de proyecto cobra una gran importancia para la ingeniería de Sistemas, pues es una frontera del conocimiento con demasiados posibles usos y aplicaciones. De igual manera, se presenta como una importante herramienta en áreas del conocimiento como biología, medicina y neurología, entre otras, pues es una forma no invasiva en que se puede comprender mejor el funcionamiento del cerebro y la forma como aprenden los seres humanos.

OBJETIVOS

GENERAL

- Implementar una librería de Software que pueda ser utilizada por los ingenieros de TECNOFACTOR, para que puedan involucrar el uso de Redes Neuronales dentro de sus aplicaciones y así desarrollar soluciones que les permitan realizar el reconocimiento de patrones en documentos a través de técnicas de Machine Learning.

ESPECÍFICOS

- Definir y desarrollar una librería en Java para la creación y uso de redes neuronales, que pueda ser escalable y fácilmente reutilizable.
- Elaborar la documentación técnica necesaria para la correcta utilización de la librería.
- Usar la librería desarrollada para definir una Red Neuronal cuyo objetivo sea el reconocimiento de dígitos escritos a mano.
- Entrenar la red neuronal definida, utilizando la información de la base de datos de dígitos escritos a mano MNIST, medir el desempeño de la red implementada y analizar los resultados obtenidos.

ALCANCE

El presente proyecto de grado tiene el siguiente alcance:

- Definición, diseño e implementación de una librería en Java, para la creación y utilización de Redes Neuronales. La librería desarrollada debe poder utilizarse para la solución de diversos problemas y funcionalidades. Debe incluir una documentación que pueda ser de utilidad para un desarrollador en el lenguaje de programación Java (JavaDoc).
- Definición e implementación de una Red Neuronal a partir de la librería desarrollada, que tenga como función el reconocimiento de dígitos escritos a mano, a partir de una imagen.
- Implementación de una funcionalidad para el tratamiento de las imágenes provenientes de la base de datos MNIST (LeCun, Corina, & J.C. Burges, s.f.), para que puedan leerse imágenes desde la base de datos y que sea de utilidad para el entrenamiento de la Red Neuronal implementada para el reconocimiento de dígitos.
- También se debe realizar el entrenamiento correspondiente de la red, y el análisis del rendimiento obtenido en el reconocimiento de dígitos.

MARCO TEÓRICO

"Existe actualmente una tendencia a establecer un nuevo campo de las ciencias de la computación que integre los diferentes métodos de resolución de problemas que no pueden ser descritos fácilmente mediante el enfoque algorítmico tradicional. Todos estos métodos se originan, de una u otra forma, en la emulación más o menos inteligente del comportamiento de los sistemas biológicos" (Espinoza & del Rosario).

De acuerdo con la definición de Red Neuronal, expresada por Simon Haykin (Haykin, 1999):

"Una Red Neuronal es un procesador distribuido, masivamente paralelo, compuesto de unidades de procesamiento simples, la cual tiene una tendencia natural para almacenar conocimiento experiencial y ponerlo a disposición para su uso. Se asemeja al cerebro en dos aspectos:

1. El conocimiento es adquirido por la red desde su entorno a través de un proceso de aprendizaje.
2. Las fuerzas de las conexiones entre neuronas, conocidas como pesos sinápticos, son usadas para almacenar el conocimiento adquirido."

Entre los beneficios que se pueden obtener con modelos computacionales basados en redes neuronales, se pueden encontrar los siguientes:

1. ***Respuesta Evidencial***: En el contexto de clasificación de patrones, una red neuronal puede ser diseñada para proveer información, no sólo sobre qué patrón particular *seleccionar*, sino también sobre la *confianza* de la selección realizada. Esta información

puede ser utilizada para rechazar patrones ambiguos, si existen, y de este modo mejorar el desempeño de la red (Haykin, 1999).

2. ***Información Contextual***: El conocimiento es representado por la propia estructura y estado de activación de una red neuronal. Cada neurona en la red es afectada potencialmente por la actividad global de todas las otras neuronas en la red. Consecuentemente, la información contextual es tratada con naturalidad por la red neuronal (Haykin, 1999).
3. ***Tolerancia a Fallos***: Una red neuronal, implementada en forma de hardware, tiene el potencial para ser inherentemente *tolerante a fallos*, o capaz de computación robusta, en el sentido de que su desempeño se degrada tenuemente bajo condiciones operacionales adversas. Por ejemplo, si una neurona o sus enlaces son dañados, la calidad de su patrón almacenado baja. Pero, dada la naturaleza distribuida de la información almacenada en la red el daño debe ser extensivo antes de que la respuesta global de la red sea degradada seriamente. Así, en principio, una red neuronal exhibe una degradación tenue en el desempeño, en vez de fallas catastróficas. Existe alguna evidencia empírica de computación robusta, pero usualmente no es controlada. Para poder asegurar que la red neuronal es de hecho tolerante a daños, puede ser necesario tomar medidas correctivas en el diseño del algoritmo utilizado para entrenar la red (Haykin, 1999).
4. ***Analogía Neurobiológica***: El diseño de una red neuronal es motivado por la analogía con el cerebro, el cual es una prueba viviente de que el procesamiento paralelo tolerante a fallos no solamente es físicamente posible, sino que es también veloz y poderoso. Los neurobiólogos utilizan las redes neuronales como una herramienta de investigación para la interpretación de fenómenos neurobiológicos (Haykin, 1999).

5. ***Adaptabilidad***: Las redes neuronales tienen una capacidad intrínseca para *adaptar* sus pesos sinápticos a los cambios en el entorno.
6. ***Plasticidad***: La plasticidad le permite al sistema nervioso en desarrollo adaptarse a su entorno. En un cerebro adulto, la plasticidad puede ser explicada por dos mecanismos: la creación de nuevas conexiones sinápticas entre neuronas, y la modificación de sinapsis existentes (Haykin, 1999).

ARQUIITECTURA DE UNA RED NEURONAL

1. Redes FeedForward de una sola capa

El tipo más simple de red neuronal es la de una sola capa, la cual consiste en un conjunto de nodos o neuronas de salida. Las entradas son llevadas directamente a las salidas a través de una serie de pesos. De esta forma, se puede considerar como el tipo más simple de una red con alimentación hacia adelante o FeedForward. Este tipo de red solamente es capaz de aprender patrones linealmente separables. (Karn, 2016).

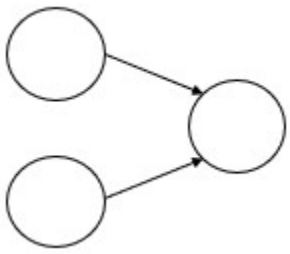

Figura 1 *Modelo de Red de una sola capa (Tchircoff, 2017)*

Redes FeedForward estándar con sólo una capa oculta pueden aproximar cualquier función continua uniformemente en cualquier rango compacto, y cualquier función medible hasta un grado deseado de precisión (Malinova, Golev, Lliev, & Kyurkchiev, 2017).

2. Redes FeedForward de múltiples capas

Una de las arquitecturas más utilizadas de redes neuronales es FeedForward, en donde la red neuronal es definida en diferentes capas o grupos paralelos, las entradas provienen de la capa inmediatamente anterior, y no es permitida la retroalimentación.

Las redes neuronales FeedForward (o con alimentación hacia adelante), son las primeras y más simples que fueron ideadas. Contienen múltiples neuronas (o nodos), distribuidas en capas. Los nodos de capas adyacentes tienen conexones entre ellas. Todas estas conexiones tienen un peso asociado (Karn, 2016).

En este tipo de red neuronal, las conexiones entre las neuronas no deben conformar ciclos o bucles, a diferencia de las redes neuronales recurrentes. La información se mueve en una sola dirección hacia adelante, desde los nodos de entrada, a través de las neuronas intermedias (si existen), hasta los nodos de salida (Feedforward neural network).

Se definen tres tipos de capas o conjuntos de neuronas:

- ***Capa de entrada*:** Los nodos de la capa de entrada proveen información desde el mundo exterior hacia el interior de la red neuronal. No se realiza ningún cálculo en ninguno de

los nodos de entrada; sólo se envía la información a las neuronas de las capas ocultas (Karn, 2016).

- ***Capas ocultas*:** Las neuronas de estas capas no tienen conexión directa con el mundo exterior (es por ello que se denominan ocultas). Realizan cálculos y transfieren la información desde la capa de entrada hasta los nodos o capa de salida. Una colección de neuronas ocultas conforma una "Capa Oculta". Mientras que una red FeedForward tiene solamente una capa de entrada y una capa de salida, puede tener cero o múltiples capas ocultas o intermedias (Karn, 2016).
- ***Capa de salida*:** Los nodos de salida colectivamente son denominados como "Capa de Salida", y son responsables de realizar cálculos y transferir la información desde la red neuronal hacia el mundo exterior (Karn, 2016).

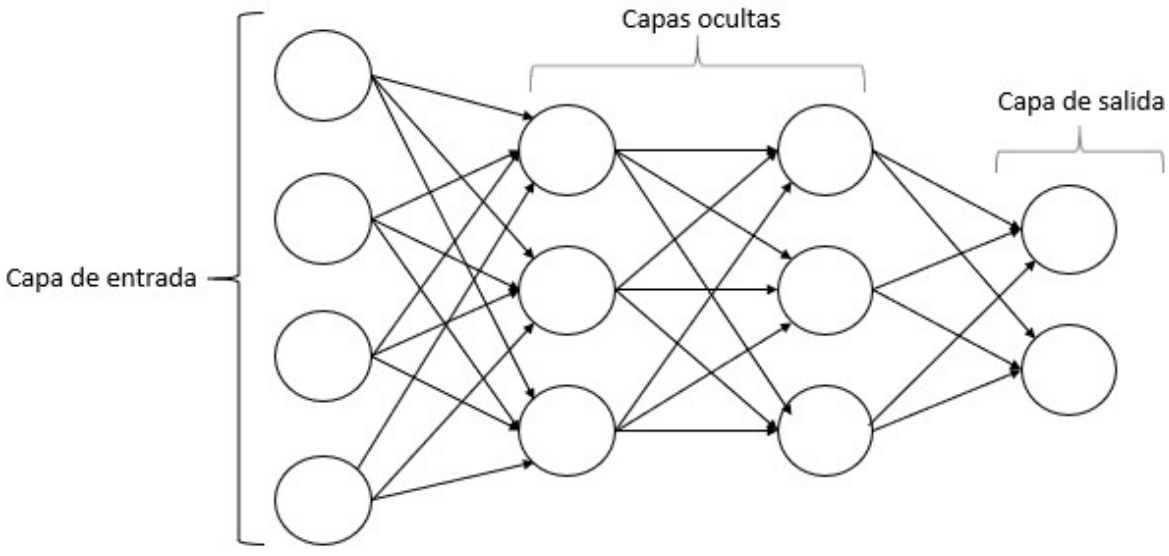

Figura 2 *Modelo de Red Neuronal FeedForward (Nielsen, s.f.)*

3. Redes recurrentes (RNN)

Una red neuronal recurrente es un tipo de red neuronal artificial en la cual las conexiones entre sus unidades forman un ciclo dirigido. Esto les permite exhibir un comportamiento temporal dinámico. A diferencia de las redes FeedForward, las redes recurrentes pueden utilizar memoria interna para procesar secuencias arbitrarias de entradas. Esto permite que sean aplicables a tareas, tales como reconocimiento del habla o de escritura a mano conectada sin segmentación (Feedforward neural network).

Este tipo de red es utilizada principalmente cuando el contexto es importante, es decir, cuando las decisiones o ejemplos de iteraciones pasadas pueden influenciar las actuales. Los ejemplos más comunes de tales contextos son los textos, donde una palabra puede ser analizada solamente en el contexto de palabras u oraciones previas (Tchircoff, 2017).

Las Redes Neuronales Recurrentes introducen un tipo diferente de neurona recurrente, en las que se recibe su propia salida con un retraso fijo de una o más iteraciones. Por lo demás, se comporta como Red FeedForward tradicional (Tchircoff, 2017).

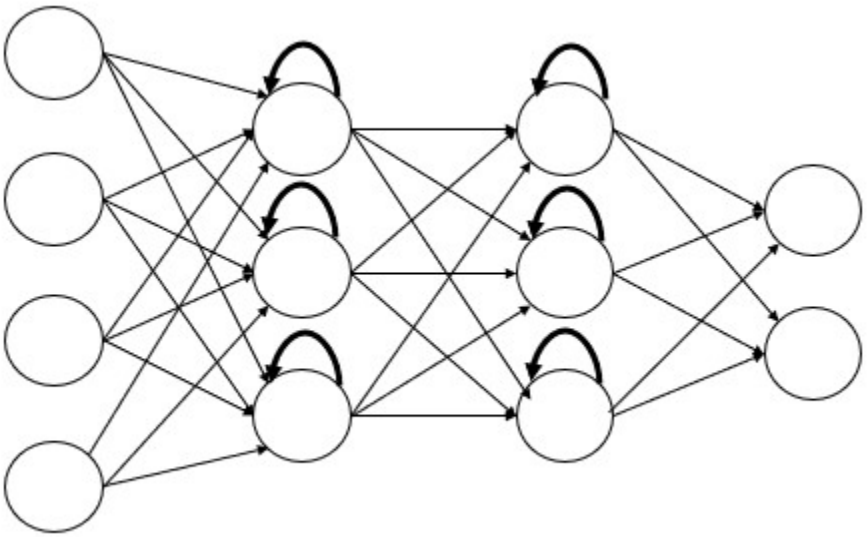

Figura 3 Modelo de Red Neuronal Recurrente (Tchircoff, 2017)

En el diagrama a continuación, se presenta una porción de la red neuronal A, que recibe la entrada X_t y genera la salida h_t. Un bucle permite que haya un flujo de información desde un paso en la ejecución de la red hacia el siguiente (Olah, 2015).

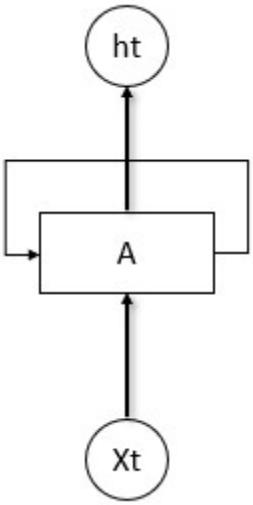

Figura 4 *Bucle de información en RNN (Olah, 2015)*

Este bucle de información hace que las redes neuronales recurrentes parezcan algo misteriosas. Sin embargo, si se observa con mayor atención, se puede apreciar que no son muy diferentes a una red neuronal normal. Una RNN puede ser representada como múltiples copias de la misma red, cada una enviando un mensaje a su sucesora. Gráficamente se puede considerar como si el bucle de información se desenvolviera (Olah, 2015).

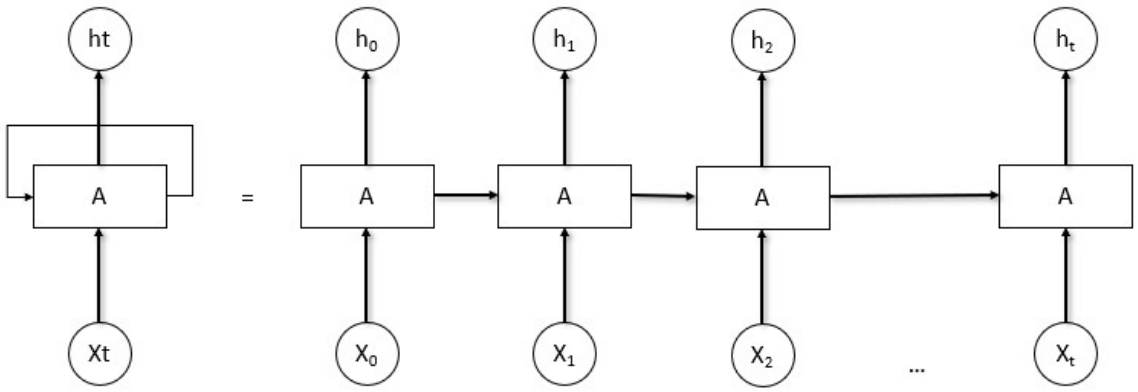

Figura 5 *Red Neuronal Recurrente desenvuelta (Olah, 2015)*

Uno de los atractivos de las RNN es la idea que pueden conectar información previa a la actual tarea, tal como usar cuadros previos de video para obtener información para la comprensión del cuadro actual. Algunas veces, sólo es necesario revisar información reciente para realizar la tarea actual. Por ejemplo, si se considera un modelo de predicción de lenguaje, intentando predecir la siguiente palabra basándose en las previas. Si se intenta predecir la última palabra en la frase "Las nubes están en el *cielo*", no es necesario tener un amplio contexto; resulta obvio que la siguiente palabra es cielo. En estos casos, donde la brecha entre la información relevante y el lugar donde es necesaria, las RNN pueden aprender a utilizar la información pasada (Olah, 2015).

Pero también hay casos donde es necesario un mayor contexto. Si consideramos el intentar predecir la última palabra en el texto "Crecí en Francia … Hablo bien el *francés*", la información reciente sugiere que la próxima palabra es probablemente el nombre de un lenguaje; pero si se desea precisar cuál, se necesita conocer el contexto de "Francia" que se encuentra mucho más atrás. Es completamente posible que la brecha entre la información relevante y el punto donde es necesaria sea muy grande. Desafortunadamente, a medida que la brecha crece, las redes neuronales recurrentes se tornan incapaces de aprender a conectar la información (Olah, 2015).

4. Redes LSTM

Deben su nombre al tipo de neurona que las constituyen. En este tipo de red neuronal se utiliza un tipo especial de neurona con memoria, que puede procesar información pasada (o con un retraso). Las redes neuronales recurrentes pueden procesar textos recordando palabras previas, y las redes LSTM pueden procesar videos recordando algo que ha sucedido muchos cuadros (o frames) atrás. También son ampliamente utilizadas en el reconocimiento de escritura y del habla (Tchircoff, 2017). Las redes LSTM son explícitamente diseñadas para evitar el problema de la dependencia a largo-plazo. Recordar información por largos periodos de tiempo es prácticamente su comportamiento genérico, y no algo que deban aprender (Olah, 2015).

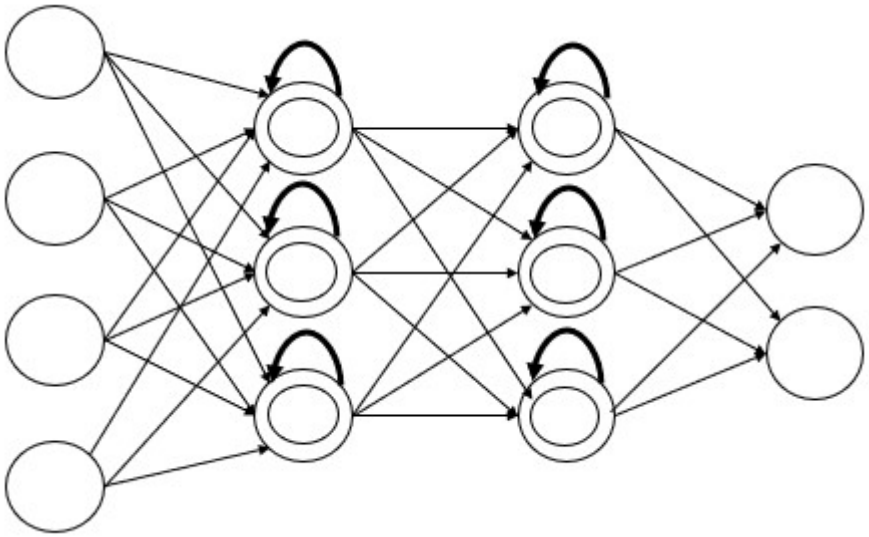

Figura 6 *Modelo de Red LSTM (Tchircoff, 2017)*

Las neuronas LSTM (Long Short-Term Memory) o de memoria a largo-corto plazo, se componen de varios elementos llamados compuertas, las cuales son recurrentes y controlan cómo la información es recordada u olvidada (Tchircoff, 2017).

Todas las redes neuronales recurrentes tienen la forma de una cadena de módulos repetidos. En las RNN estándar, dicho módulo siempre tendrá una estructura muy simple. En la figura presentada a continuación, cada línea transporta un vector de información completo, desde la salida de un nodo, hasta las entradas de otros. Los círculos representan operaciones puntuales como, por ejemplo, la suma de vectores, mientras que los rectángulos son capas de red neuronal con aprendizaje. Las líneas que se unen denotan concatenación, mientras que las líneas que se dividen significan que se envían copias de la información a los diferentes destinos. La clave principal de las redes LSTM es el estado celular, el cual es la línea que atraviesa la parte superior del diagrama (Olah, 2015).

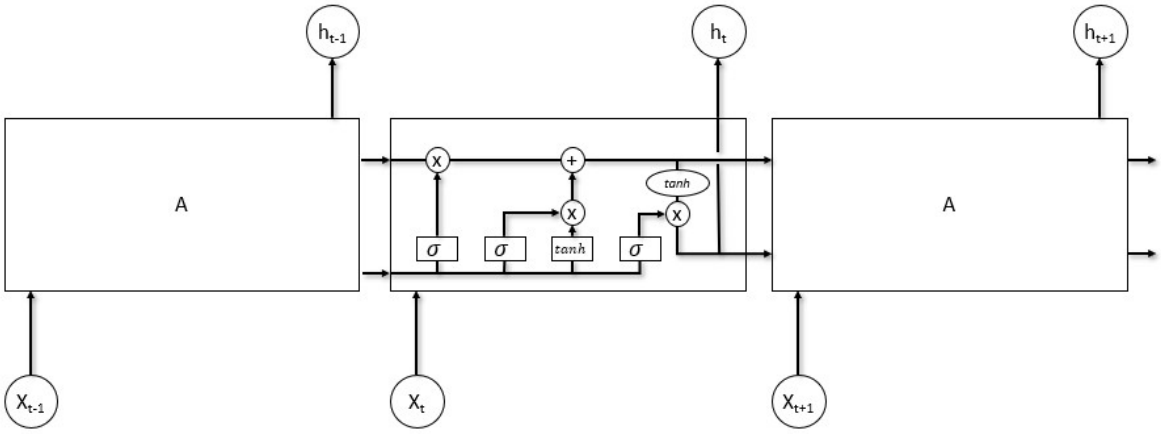

***Figura** 7 Modelo de Red LSTM (Olah, 2015)*

Puede pensarse en el estado celular como una cinta transportadora, que corre a través de la cadena completa, sólo con algunas menores interacciones lineales. Es fácil que la información fluya por este canal sin sufrir cambios. La red LSTM tiene la habilidad de añadir o remover información al estado celular, regulada cuidadosamente por las estructuras llamadas compuertas (Olah, 2015).

Las compuertas son una manera de permitir el flujo de información. Se componen de una capa de red neuronal sigmoide, y una operación de multiplicación puntual. Los números de salida de la capa sigmoide varían en un rango entre cero y uno, describiendo qué tanto de cada componente debe permitirse pasar. Un valor de cero significa que no debe pasar nada, mientras que un valor de uno significa dejar pasar todo. Un nodo LSTM tiene tres de estas compuertas, para proteger y controlar el estado celular (Olah, 2015).

5. Otras topologías de Red Neuronal

Actualmente existen muchas otras topologías de redes neuronales e inclusive cada día se pueden llegar a encontrar más (Tchircoff, 2017). Sin embargo, el detalle de estos tipos de red adicionales escapa al alcance del presente trabajo.

Para mayor información al respecto se puede consultar la lista de referencia creada por Fjodor van Veen de instituto Asimov (Tchircoff, 2017), en la que se encuentran otras topologías existentes, como por ejemplo: Gated Recurrent Unit (GRU), Auto Encoder (AE), Variational AE (VAE), Sparse AE (SAE), Deep Belief Network (DBN), Deep Convolutional Networks (DCN), entre muchas otras.

MODELO DE NEURONA

Neurona Artificial

Una neurona es una unidad de procesamiento de información la cual es fundamental para la operación de una red neuronal. Se componen de los siguientes elementos básicos:

1. *Un conjunto de sinapsis o enlaces de conexión*, cada uno de los cuales se caracteriza por tener un *peso* o *fuerza* propio. Específicamente, una señal x_j en la entrada de la sinapsis j conectada a la neurona k es multiplicada por el peso sináptico w_{kj}.
2. *Un sumador*, para realizar la adición de las señales de entrada, con el peso de su respectiva sinapsis de la neurona.
3. *Una función de activación*, para limitar la amplitud de la salida de la neurona.

4. *Un bias*, o límite, que tiene el efecto de definir el valor mínimo del resultado de la suma de las entradas multiplicadas por sus respectivos pesos, con el que se debe activar la salida de la neurona.

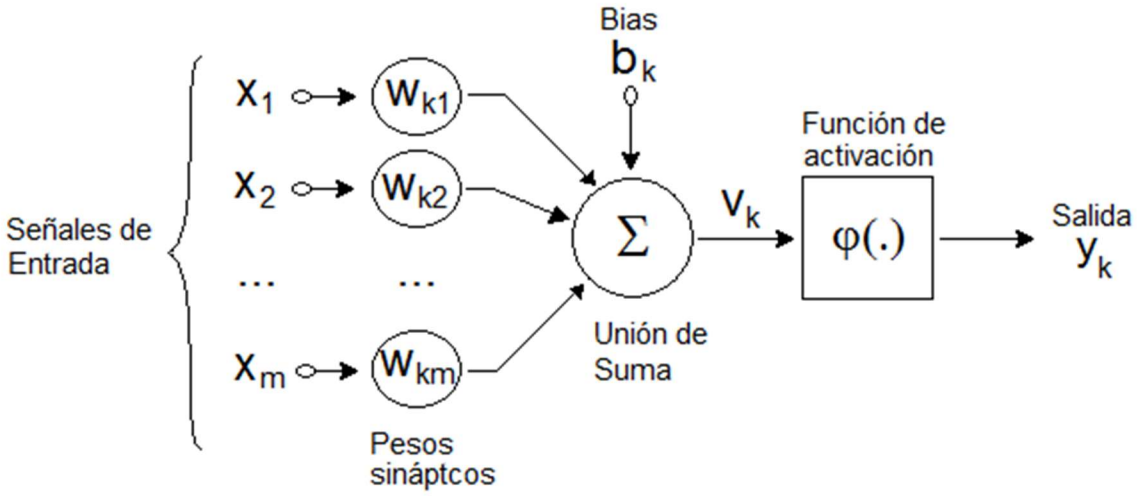

Figura 8 *Modelo de Neurona (Haykin, 1999)*

En términos matemáticos, se puede describir una neurona *k* mediante las siguientes ecuaciones:

$$u_k = \sum_{j=1}^{n} w_{kj} x_j$$

$$y_k = \varphi(u_k + b_k)$$

donde $x_1, x_2, \ldots, x_m$ son las señales de entrada; $w_{k1}, w_{k2}, \ldots, w_{km}$ son los pesos sinápticos de la neurona *k*; *u* es la combinación lineal o suma de las entradas multiplicadas por los pesos sinápticos; b_k es el valor límite, o bias, de la neurona *k*; φ es la función de activación, y y_k es la señal de salida de la neurona (Haykin, 1999).

Dependiendo del tipo de función de activación que se defina, se pueden definir diferentes tipologías de neuronas artificiales.

Función de activación

Una función de activación es un cálculo que se añade a la salida de las neuronas en una red neuronal (Sharma, 2017), y es utilizada para determinar el valor de la salida de la neurona como un Sí o un No. Mapea los valores resultantes entre -1, 0, 1, etc. dependiendo del tipo de función de activación (Sharma V., 2017). Las neuronas pueden ser clasificadas de acuerdo con el tipo de su función de activación.

Perceptrón

También conocido como modelo McCulloh-Pitts (Nielsen, s.f.), es un tipo de neurona artificial, la cual toma varias entradas $x_1, x_2, \ldots, x_n$ y produce una sola salida binaria, en la cual se utiliza una simple regla para determinar dicha salida.

La salida de la neurona (0 ó 1) es determinada a partir de la suma de los pesos de entrada; si el resultado es mayor a un valor límite b (también llamado bias), entonces la salida es 1, en caso contrario es 0. Así, la función de activación φ del perceptrón es como sigue:

$$\text{Salida} = \begin{cases} 0 \; si \sum_j w_j x_j \leq b \\ 1 \; si \sum_j w_j x_j > b \end{cases}$$

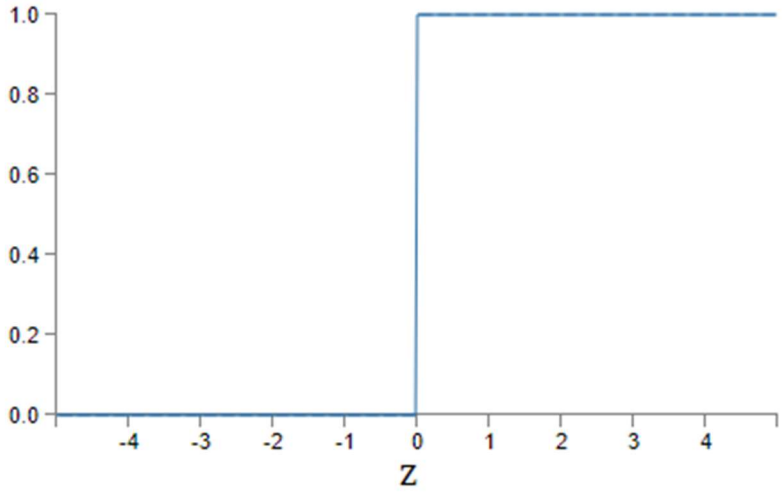

Figura 9 *Función de Activación de Neurona Perceptrón (Nielsen, s.f.)*

Sigmoide

Las neuronas Sigmoide, son similares a las Perceptrón, pero con modificaciones, cuyo propósito es obtener un cambio pequeño en la salida, si se aplican cambios pequeños en los pesos y en el bias.

La neurona sigmoide tiene entradas $x_1, x_2, \ldots, x_m$, pero en vez de tener valores de 0 ó 1, puede tomar valores reales entre 0 y 1.

Al igual que en el perceptrón, la neurona sigmoide tiene pesos definidos para las entradas $w_1, w_2, \ldots, w_m$ y un valor límite b (o bias); pero la salida no es solamente 0 ó 1, y en vez es el resultado de una función sigmoide que se da mediante la siguiente función de activación:

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

o en términos más explícitos

$$\sigma(w \cdot b) = \frac{1}{1 + e^{(-\Sigma_j w_j x_j - b)}}$$

la cual produce una función “suavizada”

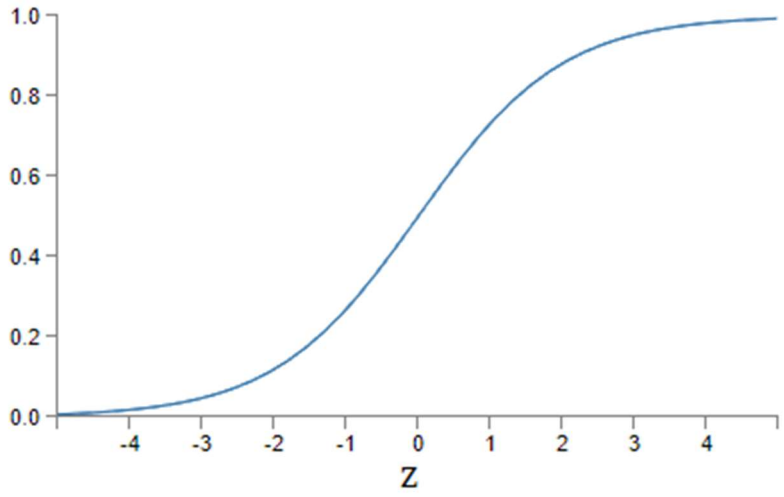

Figura 10 *Función de Activación de Neurona Sigmoide (Nielsen, s.f.)*

Una forma como se puede mejorar la calidad de los datos de salida en una red neuronal es realizando pequeños cambios en los pesos o en los bias de la red, y que estas pequeñas diferencias produzcan un cambio correspondiente en su salida. Esta propiedad es la que hace que el aprendizaje de la red sea posible.

En una red compuesta por neuronas tipo Perceptrón, dicha propiedad no es posible, puesto que un pequeño cambio en el peso de una sola entrada o en el valor límite de una neurona, puede hacer que esta cambie abruptamente su salida desde 0 a 1 o viceversa; y a su vez este cambio

puede generar el mismo efecto en otras neuronas subsecuentes, ocasionando un cambio significativo en la salida de la red. Por esta característica, se convierte en una tarea compleja poder optimizar el estado y afinar los pesos de toda la red para que produzca la salida correcta.

Es por ello que es una ventaja definir una función de activación "suavizada" para que el cambio en los pesos tenga un cambio proporcional en el resultado general de la red.

Otras funciones de Activación

Aunque la función de activación Sigmoide es una de la más ampliamente utilizada actualmente (Sharma V., 2017), existen diversas implementaciones adicionales que presentan una función continua y suavizada, definida y derivable en el intervalo posible de sus entradas y que pueden ser utilizadas en redes neuronales, por lo que se consideran otros modelos matemáticos para la implementación de redes neuronales (Malinova, Golev, Lliev, & Kyurkchiev, 2017) (Sharma V., 2017), como los siguientes:

Tangente Hiperbólica:

Otra función de activación posible es la *tanh*

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

cuya derivada es

$$tanh'(x) = \frac{1}{cosh^2 x}$$

La función tangente hiperbólica tiene características similares a la función sigmoide. Es de naturaleza no lineal, está definida en el rango (-1, 1), su gradiente es más fuerte que la función sigmoide, es decir, sus derivadas son más empinadas. Decidir entre la función sigmoide o la tangente hiperbólica dependerá de la necesidad de la fuerza del gradiente. La tangente hiperbólica es también una función de activación muy popular y ampliamente utilizada (Sharma V., 2017).

Gudermann:

La definición función de activación Gundermann es la siguiente (Malinova, Golev, Lliev, & Kyurkchiev, 2017):

$$g(x) = \int_0^x \operatorname{sech} t \, dt$$

o de manera alternativa (National Institute of Standards and Technology, 2010-2018):

$$g(x) = 2 \arctan(e^x) - \frac{1}{2}\pi = \arctan(\sinh x)$$

cuya derivada es evidentemente:

$$g'(x) = \operatorname{sech} x$$

Para gran parte del presente marco teórico se asumirá que se están utilizando neuronas tipo Sigmoide, a no ser que se especifique lo contrario. Sin embargo, las técnicas y teoría presentada son también aplicables a los demás tipos de neurona.

APRENDIZAJE Y OPTIMIZACIÓN

Función de costo

Lo que se pretende es definir un algoritmo que permita encontrar los pesos y bias adecuados dentro de toda la red para que la salida de la red se aproxime a y(x) para todas las entradas x de un conjunto de datos de entrenamiento (Nielsen, s.f.). Para cuantificar qué tan bien se está logrando dicha meta, se define la función de costo:

$$C(w,b) = \frac{1}{2n}\sum_{x}\|y(x) - a\|^2$$

Donde *w* denota a la colección de pesos en la toda la red, *b* todos los bias, *n* es el número total de valores de entrada de entrenamiento, *a* es el vector de salidas de la red cuando *x* es la entrada, *y* la suma es realizada sobre todas las entradas de entrenamiento *x*.

El objetivo del algoritmo de entrenamiento es minimizar el costo *C(w,b)* como una función de pesos y bias. En otras palabras, se pretende encontrar un conjunto de pesos y bias que haga que el costo sea el menor posible. Esto es logrado utilizando un algoritmo conocido como descenso de gradiente (Nielsen, s.f.). La forma de ejecutar la función de costo se denomina "back propagation".

Algoritmo Back Propagation

El algoritmo de propagación hacia atrás o retro-propagación (back propagation) fue introducido originalmente en la década de 1970, pero su importancia no fue apreciada hasta el paper de 1986 por David Rumelhart, Geoffrey Hinton y Ronald Williams, en el que se describen múltiples redes neuronales donde la retro-propagación funciona mucho más rápidamente que otras aproximaciones previas al aprendizaje, haciendo posible el uso de redes neuronales para resolver problemas que antes no tenían solución. Hoy en día, el algoritmo de retro-propagación es la técnica más utilizada en el aprendizaje de redes neuronales (Nielsen, s.f.).

En el corazón del algoritmo de retro-propagación se encuentra una expresión para la derivada parcial $\partial C/\partial w$ de la función de costo C con respecto a cualquier peso w (o bias b) en la red. La expresión describe qué tan rápidamente cambia el costo cuando se realizan cambios en los pesos y los biases. Así que la retro-propagación no sólo es un algoritmo veloz para el aprendizaje de las redes neuronales, sino que también provee información detallada sobre cómo los cambios en los pesos afectan el comportamiento global de la red (Nielsen, s.f.).

El algoritmo de optimización y aprendizaje, mediante la técnica de back-propagation (o retro propagación), se construye a partir de los valores de error obtenidos cuando se realiza el cálculo para una entrada a la red neuronal, conociendo el valor real esperado $e_j(n) = d_j(n) - y_j(n)$. A partir del error de las neuronas de salida, se debe obtener el valor del gradiente local para

cada neurona mediante la ecuación $\delta_j(n) = e_j(n)\ \varphi_j'\left(v_j(n)\right)$, y a partir de este definir los ajustes necesarios a los pesos $\Delta w_{ji}(n) = \ \eta.\delta_j(n).y_i(n)$.

Los valores del error en cada neurona, y los gradientes son propagados hacia atrás desde la capa de salida hacia las capas ocultas; es decir, se realiza el cálculo a partir de los valores obtenidos de las neuronas posteriores.

Ecuaciones utilizadas

Para la definición de las ecuaciones y funciones matemáticas se sigue el siguiente estándar en la numeración de los subíndices:

- Cuando se tiene un solo subíndice, se asume que hace referencia a la neurona actual. Por ejemplo u_k, hace referencia al umbral de activación *u* para la neurona *k*.
- Cuando se tienen dos subíndices, se asume que el primero hace referencia a la neurona actual, mientras que el segundo hace referencia a la neurona de la capa anterior, o que sirven de entrada a la neurona actual. Por ejemplo w_{kj}, hace referencia al peso asociado a la conexión que va desde la neurona *j* hasta la neurona *k* (típicamente entre la capa *i* e *i-1*).
- Cuando se tiene el parámetro entre paréntesis, se hace referencia a la etapa en el entrenamiento de la red neuronal. Por ejemplo, en la ecuación $w_{kj}(n+1) = w_{kj}(n+1) + \ \Delta w_{ji}(n)$, se hace el cálculo para el valor del peso entre las neuronas *j* y *k* para la iteración *n+1* del entrenamiento, a partir del cambio del peso definido en a iteración *n*.

Para obtener los valores a ajustar en los pesos en cada capa, y en el procesamiento y entrenamiento general de una red neuronal, se utilizan las ecuaciones enunciadas a continuación:

$u_k(n) = \sum_{j=1}^{m} w_{kj}(n)\, x_j(n)$	Umbral de activación de la neurona
$v_k(n) = u_k(n) + b_k(n)$	Campo local inducido
$y_k(n) = \varphi\big(u_k(n) + b_k(n)\big)$	Señal de salida de la neurona
$\varphi(z) = \frac{1}{1+e^{-z}}$	Función de activación Sigmoide
$y_k\big(v_k(n)\big) = \frac{1}{1+e^{-v_k(n)}}$	Señal de salida de la neurona Sigmoide
$e_j(n) = d_j(n) - y_j(n)$	Señal de error de la neurona de salida
$e_j(n) = \sum_k \delta_k(n) w_{kj}(n)$	Señal de error de la neurona oculta
$\delta_j(n) = e_j(n)\, \varphi_j'\left(v_j(n)\right)$	Gradiente local para neuronas de salida
$\delta_j(n) = \varphi_j'\left(v_j(n)\right) \sum_k \delta_k(n) w_{kj}(n)$	Gradiente local para neuronas ocultas
$\Delta w_{ji}(n) = \eta . \delta_j(n) . y_i(n)$	Ajuste al peso entre neuronas *i* y *j*
$\Delta b_j(n) = \eta . \delta_j(n)$	Ajuste al bias de la neurona *j*
η	Constante “Taza de aprendizaje”
$w_{kj}(n+1) = w_{kj}(n+1) + \Delta w_{ji}(n)$	Nuevo peso entre neuronas *k* y *j*

$\varphi_j'\left(v_j(n)\right) = y_j(n)[1 - y_j(n)]$ Derivada de función Sigmoide

Las etapas del entrenamiento para una Red Neuronal son las siguientes:

1. Calcular el resultado de la red

   Para cada neurona desde inicio hasta fin

   Calcular $y_k(n) = \varphi\big(u_k(n) + b_k(n)\big)$

2. Calcular la función de costo para determinar el límite del entrenamiento

   $C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a\|^2$

3. Para cada neurona desde la capa de salida hacia atrás

   Calcular el error $e_j(n) = d_j(n) - y_j(n)$ para cada neurona

   Calcular el gradiente local $\delta_j(n) = e_j(n)\,\varphi_j'\left(v_j(n)\right)$

4. Para cada neurona calcular el cambio en los pesos

   $w_{kj}(n+1) = w_{kj}(n+1) + \Delta w_{ji}(n)$

El pseudocódigo a alto nivel, del algoritmo de entrenamiento es el siguiente:

```
Para cada arreglo de entrada n
      Calcular salida de la red neuronal // FeedForward
      Para cada neurona j en la capa de salida
            Calcular señal de error y gradiente (e_j(n); δ_j(n))
      Fin para
      //Retro-propagación:
      Para cada capa oculta l desde L-1 hasta 1
            Para cada neurona j en capa l
```

```
                    Calcular gradiente (δ_j(n))
                    Calcular nuevos pesos (w_kj(n + 1))
                Fin para
            Fin para
        Fin para
Calcular el error de la salida C(w,b)
```

DÍGITOS ESCRITOS A MANO MNIST

Figura 11 *Ejemplo de dígitos escritos a mano del MNIST*

La base de datos de dígitos escritos a mano MNIST (Modified National Institute of Standards and Technology) tiene un conjunto entrenamiento de 60.000 ejemplos, y un conjunto de prueba de 10.000. Es un subconjunto de una colección mayor disponible del NIST. Los dígitos han sido escalados y centrados a un tamaño fijo de imagen (LeCun, Corina, & J.C. Burges, s.f.).

Muchos estudios dividen el conjunto de entrenamiento en dos subconjuntos que consisten en 50.000 imágenes para el entrenamiento y 10.000 para validación (Ciresan, Meier, Gambardella, & Schmidhuber, 2010).

Es una buena base de datos para aquellos que desean intentar aprender técnicas de aprendizaje y métodos de reconocimiento de patrones en datos extraídos del mundo real, y desean invertir el mínimo esfuerzo posible realizando preprocesamiento y formateo (LeCun, Corina, & J.C. Burges, s.f.).

Las imágenes originales del NIST (National Institute of Standards and Technology) fueron escaladas y normalizadas para ajustarse a un cuadro de 20x20 píxeles preservando su aspecto. Las imágenes resultantes contienen niveles de gris, como resultado de la técnica de anti-aliasing utilizada por el algoritmo de normalización. Las imágenes fueron centradas en una imagen de 28x28 calculando el centro de masa de los píxeles, y transportando la posición de la imagen para ubicar este punto en el centro del cuadro de 28x28.

La información es almacenada en un formato de archivo muy simple diseñado para almacenar vectores y matrices multidimensionales (LeCun, Corina, & J.C. Burges, s.f.).

Hay 4 archivos disponibles:

- train-images-idx3-ubyte.gz: training set images (9912422 bytes)
- train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
- t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
- t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

DISEÑO METODOLÓGICO

La metodología utilizada para la elaboración del trabajo fue la siguiente:

Etapa 1: Investigación teórica

- Propósito: Investigar la teoría y el estado del arte relativo a la construcción y el desarrollo de Redes Neuronales. Análisis de modelos matemáticos necesarios para la construcción de redes neuronales.
- Resultado esperado: Algoritmos y funciones matemáticas necesarias para la construcción de la librería.
- Metodología de trabajo: Consultas bibliográficas, lectura e investigación.

Etapa 2: Análisis y diseño de la librería a implementar

- Propósito: Definir las funcionalidades básicas que debe implementar la librería. Realizar el análisis y el diseño de la librería a desarrollar.
- Resultado esperado: Diagrama de clases detallado con las clases e interfaces a implementar. Documento con las definiciones de las funcionalidades esperadas de la librería. Algoritmos y pseudocódigos iniciales de las funcionalidades a desarrollar.
- Metodología de trabajo: Levantamiento de Requisitos, modelado con UML.

Etapa 3: Desarrollo de la librería

- Propósito: Realizar la codificación de la librería y sus interfaces. Ejecutar pruebas unitarias al código desarrollado. Elaborar la documentación técnica de la librería.
- Resultado esperado: Librería con funcionalidades para manejo de redes neuronales implementada; código fuente desarrollado. Documentación funcional.
- Metodología de trabajo: Ciclo de desarrollo iterativo para la implementación de la librería.

Etapa 4: Implementación de Red Neuronal

- Propósito: Definición inicial de una red neuronal para la identificación de dígitos escritos a mano, utilizando la librería. Implementación de interfaz para leer la base de datos de dígitos escritos a mano MNIST, para el entrenamiento de la red neuronal. Diseño de experimentos y entrenamiento de la red neuronal con base en el paper "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition" (Ciresan, Meier, Gambardella, & Schmidhuber, 2010). Definición de métricas y análisis de resultados del entrenamiento de la red neuronal.
- Resultado esperado: Red Neuronal para reconocimiento de dígitos escritos a mano.
- Metodología de trabajo: Desarrollo iterativo y pruebas de integración.

DESARROLLO DEL TRABAJO

REQUISITOS FUNCIONALES

- Se requiere crear una librería de Software que pueda ser utilizada para crear, configurar, entrenar y utilizar redes neuronales.
- La librería debe permitir crear redes neuronales de tipo FeedForward (FF)
- La librería debe permitir definir la arquitectura de la red neuronal, de acuerdo con su estructura:
    - Cantidad de Neuronas en la capa de entrada
    - Cantidad de Neuronas en la capa de salida
    - Cantidad de Capas Ocultas
    - Cantidad de Neuronas en las capas ocultas
- La librería debe permitir configurar el tipo de neurona a utilizar en las capas intermedias u ocultas y en la capa de salida, que pueden ser:
    - Perceptrón
    - Sigmoide
    - Tangente Hiperbólica
- La librería debe permitir parametrizar las variables utilizadas para el entrenamiento de las redes neuronales, que pueden ser:
    - Conjunto de datos para entrenamiento
    - Taza de aprendizaje (valor por defecto: 0.01)
    - Épocas o repeticiones

    - Error máximo esperado
    - Precisión mínima esperada

- La librería debe incluir una funcionalidad para "exportar" una Red Neuronal en un momento dado, y que se almacene su estado (es decir, los pesos y bias para cada neurona y sus sinapsis).
- La librería debe incluir una funcionalidad para "importar" una Red Neuronal que haya sido exportada previamente, para poder utilizarla, de acuerdo con su propósito.
- Las redes neuronales creadas utilizando la librería, deben realizar los cálculos necesarios para producir una salida, a partir de un vector de valores de entrada y el estado actual de los pesos de sus conexiones.
- Las redes neuronales creadas y definidas a partir de la librería, deben incluir una funcionalidad para realizar su entrenamiento y aprendizaje, utilizando las características para el entrenamiento que hayan sido definidas para la red, y a partir de la base de datos que se utilice como entrada en el entrenamiento (datos de entrenamiento).
- La librería debe permitir reportar las estadísticas del rendimiento de la red neuronal durante el entrenamiento de la red, como las épocas transcurridas, el error y la precisión obtenidos.
- El código de la librería debe ser escalable y mantenible: Debe diseñarse de manera que sea fácil la inclusión futura de nuevos tipos de red, por ejemplo, Red Recurrente y LSTM, y de neuronas, como Neurona Gudermann, LTSM, entre otros.
- Debe proveerse la documentación técnica, clara y suficiente, de las funciones y el código implementados en la librería, de forma que pueda ser consultada fácilmente por desarrolladores en el lenguaje Java.

DIAGRAMA DE CLASES

El diseño básico de la librería radica en intentar simular de la mejor manera la estructura propia de una neurona, y a partir de un conjunto de neuronas definir cada una de las posibles arquitecturas de red neuronal.

En este sentido, se define inicialmente la clase abstracta Neuron, la cual define un comportamiento básico y común para posibles tipologías de neuronas, y a partir de la cual se pueden definir sub-clases que implementen el comportamiento de la función de activación particular para cada tipo de neurona.

Con la intención de simular más aproximadamente la estructura y el funcionamiento de una red neuronal biológica, se diseña la estructura de una neurona para que contenga un axón, en el que se manifiesta la salida de la neurona, y un conjunto de dendritas, que se encargan de obtener los valores de entrada para la neurona a partir de axones de neuronas precedentes.

Por ello se implementan las clases Dendrite y Axon. Cada neurona contiene un Axon y un conjunto de Dendrite. El Axon se encarga de calcular el valor de salida de la neurona, mediante el método *synapse*, que obtiene el valor de la función de activación de la neurona. Cada Dendrite se encarga de obtener un valor de entrada, mediante el método *syapse*, que obtiene el valor almacenado en el axón de la neurona precedente a la cual está asociada la dendrita.

A partir de la clase Neuron se definen diferentes sub-tipos de neurona, cada una con la implementación del método que calcula la función de activación particular para su tipo. Por ejemplo, la clase Sigmoid, es un tipo de Neuron, cuyo método “activationFunction” calcula la

función Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$. De igual manera, se implementa la derivada apropiada para cada tipo de función de activación.

Los tipos de Neuron implementados en el proyecto, son, Sensor, Perceptron, Sigmoid, Gudermann, HyperbolicTangent.

Para facilitar la generación de redes neuronales y la creación de neuronas, se implementa el patrón de diseño Factory, mediante la clase NeuronFactory.

La definición de las redes neuronales se realiza mediante la clase abstracta NeuralNetwork, la cual define un comportamiento básico y común para posibles tipologías de redes neuronales y además contiene el conjunto de neuronas que integran la red, distribuidas en capas, de acuerdo con la arquitectura de red neuronal que se utilice. A partir de esta se pueden definir sub-clases que implementen el comportamiento particular para cada tipología de red neuronal.

A partir de la clase NeuralNetwork, se define la sub-clase FeedForwardNetwork, con la implementación necesaria para una red neuronal tipo FeedForward. Por ejemplo, en ésta se implementa el método *createLayers*, en el cual se definen las capas de la red neuronal, cuyas neuronas en cada capa, transmiten siempre la información hacia adelante, hacia las neuronas de la capa siguiente (las dendritas de las neuronas de la capa *i* siempre estarán conectadas a los axones de las neuronas de la capa *i-1*) sin que se presenten bucles en el flujo de la información.

Para facilitar la generación de redes neuronales, se implementa el patrón de diseño Factory, mediante la clase NeuralNetworkFactory.

Se realiza sobrecarga en la definición de los métodos de creación de redes neuronales para poder definir características de la red, como cantidad de neuronas en la capa de entrada, cantidad

de neuronas en la capa de salida, cantidad de capas ocultas, cantidad de neuronas en las capas ocultas, tipo de red, tipo de neuronas a utilizar, entro otros parámetros para la definición de la red.

En la clase NeuralNetworkFactory también se implementan los métodos para realizar la exportación y la importación de las redes neuronales, hacia y desde archivos xml respectivamente.

Se define la interfaz DataSet para especificar el contrato que debe cumplir cualquier implementación que haga uso de la API para realizar la lectura de los datos que se utilizarán tanto para el entrenamiento como para las pruebas de la red neuronal. Se debe utilizar también una implementación del DataSet para transferir los datos que serán utilizados por la red neuronal ya entrenada, para su funcionamiento.

A continuación, se presenta un diagrama de clases resumido, en el que se define el diseño básico de las clases que conforman la librería "*NeuralNetworksAPI*".

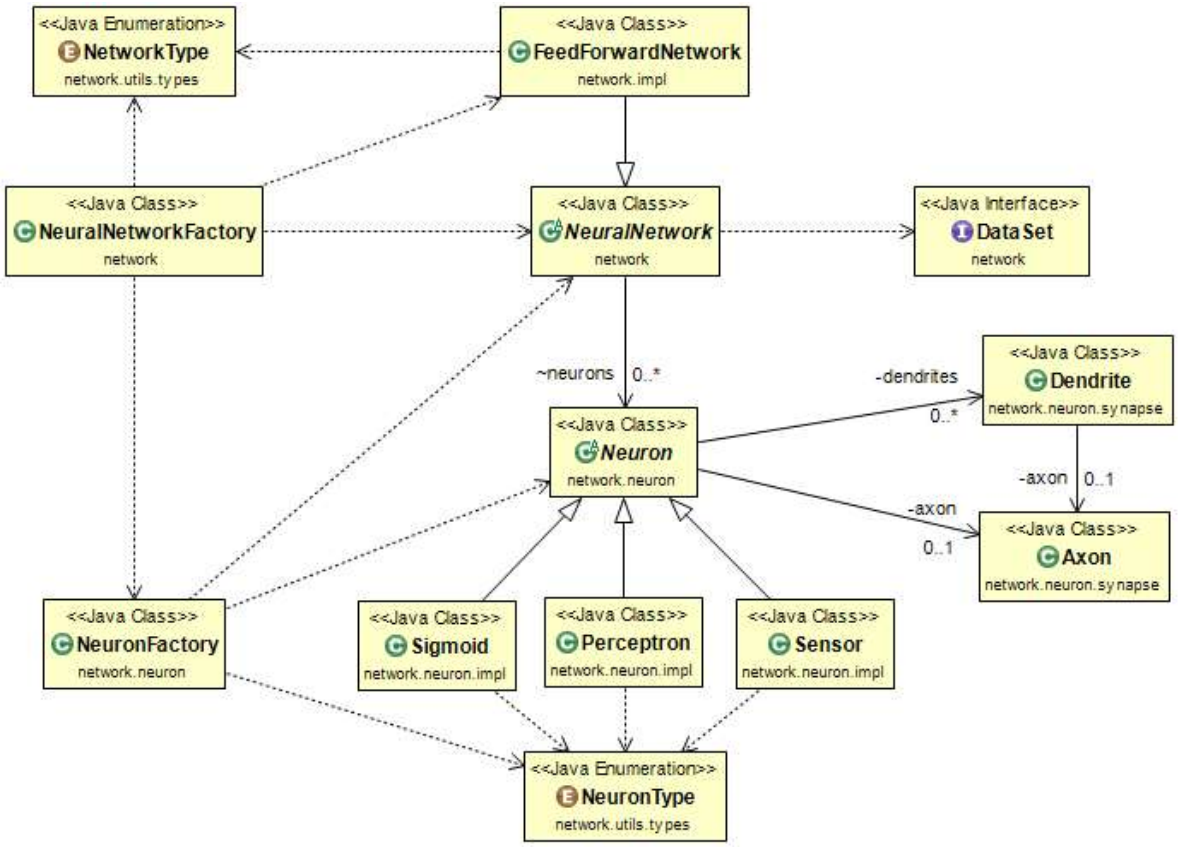

Figura 12 *Diagrama de clases básico de la librería*

A continuación, se presenta un diagrama de clases resumido, en el que se define el diseño básico de las clases que conforman la implementación e interacción del proyecto de prueba con la librería *NeuralNetworksAPI*, para realizar la lectura de las imágenes de la base de datos MNIST.

Para enviar las imágenes de los dígitos de la base de datos MNIST, tanto de entrenamiento como de prueba, se realiza una implementación de la interfaz DataSet, para que pueda ser utilizada por los métodos de la librería. Dicha implementación se realiza en la clase MNISTDataSet, la cual se encarga de leer los archivos *t10k-images.idx3-ubyte*, *t10k-labels.idx1-ubyte*, *train-images.idx3-ubyte* y *train-labels.idx1-ubyte*, interpretarlos como un objeto de tipo

MNISTImage para su fácil procesamiento, y tenerlos disponibles para que puedan ser accedidos por los métodos en las clases de la API.

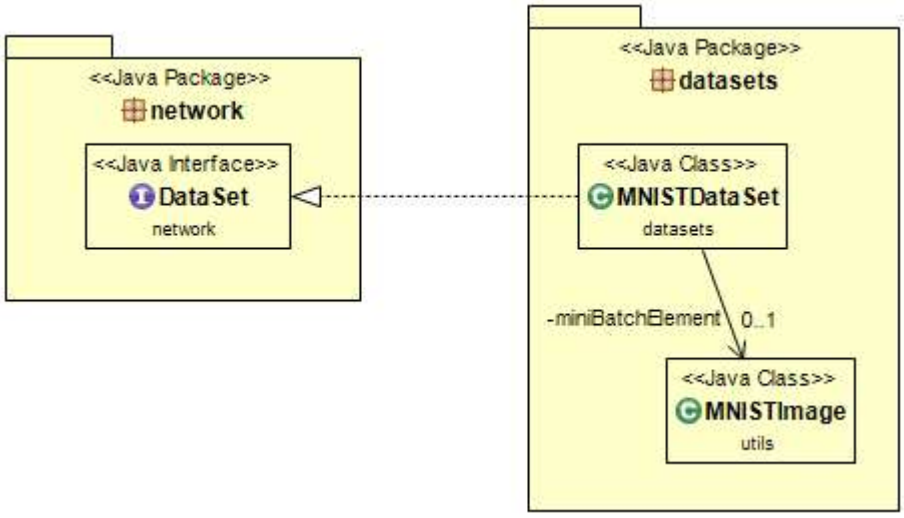

Figura 13 *Diagrama de clases de uso de la librería*

ALGORITMOS Y PSEUDO CÓDIGO

A continuación, se detallan los algoritmos utilizados para las funcionalidades principales desarrolladas en la librería. En particular, se detallan los algoritmos utilizados para los métodos de procesamiento y de entrenamiento de las redes neuronales.

Algoritmos de procesamiento

El propósito de los métodos de procesamiento es el de calcular la salida de la red neuronal; para ello, se define un comportamiento que puede ser generalizado para múltiples arquitecturas de red neuronal. Se realiza una iteración sobre todas las capas de la red neuronal, para que cada una de las neuronas de la red calcule el valor de su salida.

Se incluye un control con un número de identificador de proceso, para que una neurona pueda identificar si todas sus neuronas de entrada ya han calculado la salida correspondiente al identificador de proceso en ejecución; y así evitar calcular una salida, cuando todas las entradas aún no se encuentran disponibles. Dicho control puede resultar útil para implementaciones de Redes Neuronales Recurrentes (RNN).

```
procesar(entrada)
	guardar entrada en neuronas de capa de entrada
	procesar()
fin
```

```
procesar()
	id_proceso = generar un id de proceso

	mientras que neuronas_pendientes_por_procesar
		neuronas_pendientes_por_procesar = falso
		para cada una de las capas de la red neuronal
			para cada una de las neuronas de la capa
				neurona_procesada = neurona.procesar(id_proceso)
				si neurona_procesada = falso
					neuronas_pendientes_por_procesar = verdadero
				fin si
			fin para
		fin para
	fin mientras
	retornar valores de las neuronas de la capa de salida
fin
```

En particular, para una red FeedForward se simplifica el algoritmo anterior, dado que no se requiere el control con el identificador de proceso, así:

```
procesar()
	para cada una de las capas de la red neuronal
		para cada una de las neuronas de la capa
			neurona.procesar()
		fin para
	fin para
	retornar valores de las neuronas de la capa de salida
fin
```

El algoritmo para el procesamiento (y el cálculo del valor de salida) dentro la neurona, es el siguiente:

```
neurona.procesar(id_proceso)
	si se ha procesado previamente la Neurona para el id_proceso
		retornar verdadero
	fin si

	entradasListas = verdadero
	//Se valida si todos los axones de entrada, asociados a las dendritas
	//tienen igual código de proceso,
	//para garantizar que ya hayan realizado el cálculo correcto:
	//dendrites.axon.id_proceso = id_proceso
	para cada dendrita d
		si d.axon.id_proceso = id_proceso
			d.sinapsis()
		de otro modo
			entradas_listas = falso
		fin si
	fin para

	si entradasListas es verdadero
		axon.sinapsis(id_proceso)
		retornar verdadero
	fin si
	retornar falso
fin
```

La implementación para redes FeedForward simplifica el algoritmo anterior, de la siguiente manera:

```
neurona.procesar()
	para cada dendrita d
		d.sinapsis()
	fin para
	axon.sinapsis();
fin
```

Los métodos sinapsis (diseñados para emular el comportamiento de un cerebro biológico) son los que se encargan de ejecutar la comunicación entre las neuronas.

Cuando se realiza la activación de la sinapsis en las dendritas, se lee el valor de la salida de la neurona a la que se encuentra conectada la dendrita a través del axón de ésta.

Cuando se realiza la sinapsis en el axón, se llama la función de activación de la neurona, para que se realice su cálculo correspondiente, y el valor resultante es almacenado en el axón. De esta manera, todas las dendritas que se encuentran conectadas al axón pueden obtener el valor de la salida de la neurona cuando se activen sus sinapsis, sin que haya necesidad de volver a realizar el cálculo de la función de activación.

Para la dendrita, se define el método sinapsis de la siguiente manera:

```
dendrita.sinapsis()
	dendrita.entrada = axon.salida;
fin
```

Para el axón, se define de la siguiente manera:

```
axon.sinapsis(id_proceso)
	axon.id_proceso = id_proceso
	axon.salida = neurona.funcion_de_activacion()
fin

axon.sinapsis()
	axon.salida = neurona.funcion_de_activacion()
fin
```

El algoritmo para la función de activación depende del tipo de neurona implementado, y varía principalmente en la función matemática asociada al tipo de neurona. Por ejemplo, para las neuronas tipo Sigmoid, se definen los siguientes métodos:

```
//para neurona sigmoid
neurona.funcion_de_activacion()
	//se define la función sigmoide a partir de
	//las entradas y los pesos de las dendritas de la neurona
	retornar 1/(1+e^(-Σ_j w_j x_j - b))
fin

neurona.derivada_funcion_activacion()
	y = funcion_de_activacion()
	//se realiza
	retornar (y * (1 - y))
fin
```

El siguiente es el método definido para calcular el campo local en cada neurona; es decir, la implementación algorítmica de la sumatoria de los pesos de cada dendrita, multiplicados por el valor de la salida de la neurona anterior. En expresión matemática $v_k(n) = u_k(n) + b_k(n)$; o en otros términos $\Sigma_j w_j x_j + b$, para ser utilizado más fácilmente en cada función de activación, independiente del tipo de neurona implementado.

```
neurona.campo_local()
	resultado = 0
	//para cada una de las entradas, se realiza la multiplicación
	//del peso por el valor de la entrada, y se suma el resultado
	para cada dendrita de la neurona
		resultado = resultado + (dendrita.peso * dendrita.entrada)
	fin para
	resultado = resultado + bias
```

```
        retornar resultado
fin
```

Algoritmos de entrenamiento

A continuación, se presentan los algoritmos que serán implementados en el entrenamiento de las redes neuronales.

El siguiente es el método utilizado para realizar el entrenamiento de la red neuronal a partir de un set de datos. El entrenamiento se realiza utilizando subconjuntos (o mini-batches) del conjunto total de datos, con el objetivo de mejorar el rendimiento del entrenamiento, y solamente ajustando los pesos de las conexiones al final de la ejecución de cada subconjunto. El tamaño del subconjunto puede variar entre 1 y el tamaño total del conjunto de datos.

El entrenamiento se limita a partir de algunos parámetros que indican los niveles de tolerancia o la cantidad de repeticiones esperadas. El parámetro de épocas para el entrenamiento define la cantidad de veces o repeticiones que se ejecutará sobre el conjunto total de datos.

Los valores de error máximo y de precisión mínima definen el error aceptado y el mínimo de precisión esperada del entrenamiento. El método se ejecuta hasta alcanzar cualquiera de estos valores.

```
entrenar(datos, epocas, max_error, min_precision)
        //iniciar el error en un valor mayor al máximo permitido
        error = max_error + 1
        precision = 0
        i = 0
```

```
//se valida el número de épocas, el error, y la precisión
mientras que (
       i < epocas Y
       error > max_error Y
       precision < min_precision) haga
       i = i + 1
       error = 0
       precision = 0
       //se inicia el contador de casos de entrenamiento
       n = 0

       //se preparar los sub-conjuntos de etrenamiento
       //lo ideal es que los elementos de cada subconjunto
       //sean definidos aleatoriamente
       datos.preparar_subconjuntos()

       //para cada sub-conjunto
       //se debe procesar el entrenamiento y calcular el error
       para cada uno de los subconjuntos
              para cada registro en el sub-conjunto
                     entrada = datos.obtener_proxima_entrada()
                     salida_esperada =
                     datos.obtener_proxima_salida_esperada()

                     //ejecutar el entrenamiento de la Red
                     //para un registro de prueba
                     salida = entrenar(entrada, saida_esperada)

                     //se utiliza la salida obtenida
                     //para medir la precisión y el error

                     si salida coincide con salida_esperada
                            precision = precision + 1
                     fin si
                     error = ∑_x ||y(x) − a||^2
```

```
                        n = n + 1
                fin para

                para cada capa
                        para cada neurona
                                neurona.ajustar_pesos()
                        fin para
                fin para

        fin para

        //calcular el error promedio y la precisión
        error = error / 2*n
        precision = precision / n;

    fin mientras
fin
```

El siguiente es el algoritmo utilizado para el entrenamiento a partir de una sola entrada. El propósito del algoritmo es procesar el valor actual que arroja la red neuronal para una entrada específica, y a partir de su salida determinar la diferencia o el error existente, en comparación con la salida esperada.

El valor del error obtenido en la capa de salida se propaga desde la capa de salida hacia la primera de las capas ocultas, para distribuir el cambio proporcional en cada peso de cada dendrita.

Posteriormente, cada neurona se encarga de calcular los valores a ajustar en los pesos de sus conexiones, mediante los métodos de back-propagation. Se utiliza el método de retro-propagación (o back-propagation) para calcular los valores de los gradientes locales que serán

utilizados por cada neurona y por las neuronas que le preceden. Se denomina de retro-propagación, dado que estos valores se calculan a partir de las neuronas de salida, y se utilizan para las neuronas precedentes, capa por capa.

```
entrenar(entrada, salida_esperada)
       //se realiza el procesamiento normal del cálculo de la Red
       salida = procesar(entrada)
       para cada neurona de la capa de salida
              //calcular el gradiente de la Neurona y
              //acumular el ajuste a los pesos
              //se realiza dentro del método de retro-propagación
              neurona.back-propagation(saida_esperada)
       fin para

       para cada capa oculta, desde la última hasta la primera
              para cada neurona en la capa
                     //calcular el gradiente de la Neurona y
                     //acumular el ajuste a los pesos
                     //se realiza dentro del método de retro-propagación
                     neurona.back-propagation()
              fin para
       fin para
fin
```

Los métodos de retro-propagación o back-propagation son diferentes, dependiendo de si la neurona pertenece a la capa de salida o a una capa oculta. Básicamente, se diferencian en el cálculo del error que proviene de la capa posterior. Para las neuronas de salida, el error se da por $e_j(n) = d_j(n) - y_j(n)$; mientras que para las neuronas, ocultas el error se da por, $\sum_k \delta_k(n) w_{kj}(n)$. A partir de estos valores es posible calcular el Gradiente local, así:

$\delta_j(n) = e_j(n)\, \varphi_j'\left(v_j(n)\right)$; gradiente local para neuronas de salida

$\delta_j(n) = \varphi_j'\left(v_j(n)\right)\sum_k \delta_k(n)w_{kj}(n)$; gradiente local para neuronas ocultas

```
//para neurona en capa de salida
neurona.back-propagation(valor_esperado)
	neurona.propagar((-1) * neurona.error(valor_esperado))
fin

//para neurona en capa oculta
neurona.back-propagation()
	neurona.propagar(neurona.error())
fin
```

Como se mencionó anteriormente, la diferencia principal radica en la forma de calcular el error. A partir de éste, el cálculo del gradiente local se realiza de igual manera para ambos tipos de neurona.

```
//para neurona en capa de salida
neurona.error(double valor_esperado)
	//se calcular el error
	error = (valor_esperado - neurona.salida)
	retornar error
fin

//para neurona en capa oculta
neurona.error()
	//se calcula el error
	error = 0
	//se debe calcular ∑_k δ_k(n)w_kj(n)
	//Para cada una de las neuronas asociadas al Axón,
	//obtener el gradiente local multiplicado por el peso,
	//es decir, para cada conexión de salida
	para cada dendrita conectada al Axon de la neurona
		error = error +
```

```
                (dendrita.neurona.gradiente_local * dendrita.peso)
        fin para
        retornar error
fin
```

En el método propagar, se realiza el cálculo del gradiente local, que será utilizado por las neuronas de capas anteriores (en realidad este es el valor que se propaga hacia atrás), además de definir el valor del cambio en el peso que se debe aplicar en cada conexión. Dado que se realiza una definición del entrenamiento considerando sub-conjuntos de datos, los valores del cambio serán actualizados posteriormente en el método ajustar_pesos. El ajuste es realizado una vez se haya calculado el cambio ponderado, es decir, la sumatoria de los cambios para todas las entradas del sub-conjunto.

```
neurona.propagar(error)
        //se obtiene el valor del gradiente local de la Neurona,
        //a partir del error y la derivada de la función de activación
        gradiente_local = (error * neurona.derivada_funcion_activacion())

        //para cada una de las sinápsis de entrada (Dendritas)
        para cada dendrita de la neurona
                //se define el cambio en el peso a aplicar en la entrada,
                //a partir del gradiente local, el valor de entrada,
                //y la tasa de aprendizaje
                dendrita.cambio = dendrita.cambio +
                        (tasa_aprendizaje * gradiente_local * entrada)
        fin para
```

```
        //se define el cambio en el bias, a partir del gradiente local
        //y la tasa de aprendizaje
        neurona.cambio_bias = (taza_aprendizaje * gradiente_local)

fin
```

La tasa de aprendizaje es un valor que puede ser definido globalmente para el entrenamiento. Si el valor es pequeño, el entrenamiento es más confiable, pero el tiempo de entrenamiento será mayor. Si el valor es alto, es posible que el entrenamiento no converja a un estado óptimo, dado que los cambios en los pesos pueden llegar a ser tan grandes que sobrepasen el valor necesario y el estado de la red no mejore. Un valor apropiado puede ser de 0.01 (Surmenok, 2017).

El algoritmo para realizar el ajuste de los pesos en cada neurona, después de haber ejecutado un conjunto de datos de entrenamiento (incluso cuando se definen sub-conjuntos de un sólo elemento, es decir, cuando se ajusta el peso de las conexiones para cada entrada), es el siguiente:

```
neurona.ajustar_pesos()
        //para cada una de las sinapsis de entrada a la (Dendritas)
        para cada dendrita
                //se define el nuevo valor para el peso de la entrada
                dendrita.peso = (dendrita.peso - dendrita.cambio)
                dendrita.cambio = 0
        fin para

        //se define el nuevo valor para el bias de la Neurona
        neurona.bias = (neurona.bias - neurona.cambio_bias)
        neurona.cambio_bias = 0
fin
```

Algoritmos para la utilización de la librería

Una vez implementada la lectura de los datos (en este caso la base de datos MNIST), se puede hacer uso, relativamente trivial, de la librería. El trabajo del desarrollador que utilice la API debe enfocarse en realizar la traducción desde la información con la que se va a utilizar la red neuronal, implementando la interfaz DataSet. El algoritmo para utilizar la red neuronal y realizar el entrenamiento, puede ser tan sencillo como se presenta a continuación.

Se define la estructura de la red neuronal a implementar con datos como el nombre, la cantidad de neuronas en la capa de entrada, cantidad de neuronas en la capa de salida, cantidad de capas ocultas y cantidad de neuronas en las capas ocultas. Como se ha mencionado anteriormente, los parámetros para definir los límites del entrenamiento son el conjunto de datos a utilizar, la cantidad de repeticiones o épocas, el error máximo permitido y la precisión mínima esperada.

```
datos = leer_datos_entrenamiento_mnist()
red_n = libreria.crear_red("mnist", 28*28, 10, 2, 15)
red_n.entrenar(datos, 100, 0, 0.9999)
red_n.exportar("mnist")
```

Una vez la red neuronal se ha entrenado, se puede importar el archivo generado para utilizarla y probarla con el archivo de pruebas del MNIST.

```
datos = leer_datos_prueba_mnist()
red_n = libreria.importar("mnist")
//datos, epocas, max_error, min_precision
red_n.probar(datos, 100, 0, 0.9999)
```

También se puede utilizar la red de la misma manera para procesar una imagen cada vez, por ejemplo, si la lectura se hace a partir de una interface de usuario.

```
red_n = libreria.procesar(datos.proximo_valor)
```

RESULTADOS

1. Se realizó la definición y el desarrollo de la librería “NeuralNetworksAPI” en Java para la creación y uso de redes neuronales, que puede ser escalable y reutilizable. Se entrega proyecto Java, y archivo *.jar*, adjunto con el presente documento
2. Se elaboró la documentación técnica necesaria para la correcta utilización de la librería. Se entrega documentación JavaDoc del proyecto, adjunto con el presente documento.
3. Se utilizó la librería desarrollada para definir una Red Neuronal cuyo objetivo es el reconocimiento de dígitos escritos a mano. Se realizó el entrenamiento de la red neuronal definida, utilizando la información de la base de datos de dígitos escritos a mano MNIST, y se midió el desempeño de la red implementada. Se entrega, adjunto con el presente documento, archivo *.xml*, que puede ser interpretado por la librería, con una red neuronal entrenada para reconocer dígitos escritos a mano.

Los parámetros de entrenamiento de la red neuronal entregada son:

- Capas ocultas: 3
- Cantidad de neuronas en capas ocultas: 28
- Tipo de neurona utilizada: Sigmoid
- Taza de aprendizaje: 0.1
- Épocas de entrenamiento: 100

Al comparar contra el set de datos de entrenamiento del MNIST, se obtiene:

- Error: 0.026
- Precisión: 96.71%

Al comparar contra el set de datos de prueba del MNIST, se obtiene:

- Precisión: 95.18%

Entregables

Los entregables del trabajo de grado serán los siguientes:

- Diagrama de clases desarrolladas para la librería.
- Librería desarrollada en JAVA para la creación y uso de Redes Neuronales.
- Documentación técnica de la librería
- Red Neuronal para identificación de dígitos escritos a mano
- Código fuente desarrollado

CONCLUSIONES

I.

No es común encontrar implementaciones de redes neuronales que basen su funcionamiento en emular el comportamiento de neuronas individuales; es decir, que cada neurona sea un elemento independiente. En vez de esto, se encuentran múltiples modelos e implementaciones basados en álgebra lineal y modelos y cálculos matriciales, debido a que las estructuras y la formulación matemática propia de las redes neuronales se presta fácilmente para ello.

Sin embargo, en la experiencia de realizar la definición y el desarrollo de la librería, implementándola a partir de neuronas como entidades independientes, resulta más fácil de visualizar y comprender la teoría sobre redes neuronales; y puede utilizarse con un enfoque más pedagógico, al hacerse una aproximación más directa al marco teórico.

II.

Una de las dificultades que se encontraron al momento de realizar la implementación, es el cuidado que se debe tener en la programación de las fórmulas y funciones matemáticas. Aunque existe abundante documentación en relación con el funcionamiento de redes neuronales y su entrenamiento, la teoría está altamente fundamentada en modelos matemáticos. Pese a que los principios básicos son los mismos, tanto la notación como la forma de aproximarse a los fundamentos matemáticos varía en gran medida de una fuente bibliográfica a la siguiente.

En este contexto, se presentaron algunos problemas en la implementación de los algoritmos de entrenamiento, pues un signo contrario en una operación, o un error pequeño en la implementación de una fórmula matemática, hizo que el algoritmo de entrenamiento no funcionara correctamente y la red neuronal no aprendiera. Este tipo de error es particularmente difícil de encontrar y depurar, y uno en que se tuvo que invertir una muy considerable cantidad de tiempo.

Una de las operaciones en las que se encontraron mayores inconvenientes, fue la ecuación utilizada para calcular el nuevo peso a aplicar entre dos neuronas. Por ejemplo, en el libro de Haykin (Haykin, 1999), ecuación 2.4 de la página 75, se encuentra que el valor para el nuevo peso entre las neuronas j y k, es:

$$w_{kj}(n+1) = w_{kj}(n+1) - \Delta w_{ji}(n).$$

Similarmente, el texto de Michael Nielsen (Nielsen, s.f.), en el capítulo 1, establece que el nuevo peso para las conexiones se basa en las ecuaciones:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Después de realizar muchas ejecuciones de prueba, depuración detallada y revisiones del material bibliográfico, se encontró que el valor no debe ser una resta sino una adición, ya que el valor negativo del cambio en el peso se origina desde el cálculo del error, en:

$$e_j(n) = d_j(n) - y_j(n)$$

III.

Dado que las redes neuronales, por su propia naturaleza, presentan procesamiento con una gran cantidad de elementos funcionando en paralelo, y cuyos valores y pesos en las conexiones no son evidentes para el resultado final, una de las mayores dificultades al utilizar redes neuronales, es realizar el seguimiento y depuración de los valores.

Al intentar realizar los ajustes necesarios del problema enunciado en el numeral anterior, se encontró la necesidad de tener una forma de revisar el resultado de cada cálculo individual en cada una de las conexiones de las neuronas. Esto resulta en una tarea laboriosa, dado que la cantidad de conexiones en una red neuronal FeedForward crece exponencialmente con cada neurona que se incluya en el modelo; considerando que para el problema de interpretar los dígitos MNIST, tan sólo en la capa de entrada son necesarias 28x28=784 neuronas.

III.

En el entrenamiento de redes neuronales, no siempre más es mejor. Dada la gran cantidad de conexiones que pueden presentarse en una red, el tiempo de procesamiento tiende a escalar en gran medida entre mayor sea la cantidad de neuronas que contiene. Es posible “sobre entrenar” una red, y que al hacerlo pierda esta precisión.

Además, tal como se puede apreciar en (Ciresan, Meier, Gambardella, & Schmidhuber, 2010), una mayor cantidad de capas, neuronas y/o conexiones no garantiza una mejora proporcional en los resultados y la precisión de la red neuronal. De igual manera, puede no darse una ganancia en precisión justificable, dado que el tiempo de procesamiento es directamente proporcional al

número de neuronas y crece de manera exponencial entre más capas existan en la red, pues el número de conexiones entre neuronas crece exponencialmente.

IV.

La librería implementada puede ser utilizada en su estado actual, por los desarrolladores de Tecnofactor fácilmente, incluyendo el archivo *.jar* en sus proyectos. Puede también ser mejorada y ampliar su alcance para servir a múltiples proyectos de acuerdo con la necesidad que se tenga. La red neuronal incluida puede también ser utilizada dentro de un software en conjunto con otras funcionalidades, para realizar reconocimiento y procesamiento de textos.

TRABAJO FUTURO

Como continuación de este trabajo de grado, existen diversas alternativas de trabajo que se pueden continuar a partir de lo entregado. Algunas de ellas pueden ser las enunciadas a continuación:

- Realizar la implementación de más arquitecturas de red neuronal, como puede ser la inclusión en la librería para generar Redes Neuronales Recurrentes y Redes LSTM, complementando así las funcionalidades existentes y pueden brindar posibilidades adicionales para los objetivos de Tecnofactor.
- Implementar una estrategia para que las redes neuronales generadas tengan la posibilidad de continuar su entrenamiento mientras están en uso (aprendizaje no supervisado).
- Implementar funcionalidades de optimización en la ejecución, como por ejemplo la inclusión de un proceso de "poda", en que las conexiones que no son utilizadas en la red, o cuyo peso es muy cercano a 0, sean eliminadas, disminuyendo así el tiempo de procesamiento de la red.
- Entrenar una red neuronal para reconocer caracteres escritos a mano (y no solamente dígitos) y aprovecharlo en aplicaciones de carácter educativo para el aprendizaje de la lectura y escritura.
- Implementar y entrenar redes neuronales orientadas al análisis e interpretación de textos, para la creación de un identificador de documentos tipo factura, para reconocerlos de manera automática y extraer la información básica necesaria para otros procesos, como facturación electrónica, factoring y confirming, sin importar el sistema origen de la información.

REFERENCIAS

Ciresan, D., Meier, U., Gambardella, L., & Schmidhuber, J. (2010). Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition. *Neural Computation, Volume 22, Number 12*. Obtenido de Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition: https://arxiv.org/abs/1003.0358

Diaz, A., Ghaziri, H., & Glover, F. (1996). *Optimización heurística y Redes neuronales.* Madrid: Paraninfo.

Espinoza, V., & del Rosario, M. (s.f.). *Las Redes Neuronales Artificiales y su importancia como herramienta en la toma de decisiones.* Obtenido de Las Redes Neuronales Artificiales y su importancia como herramienta en la toma de decisiones: http://sisbib.unmsm.edu.pe/bibvirtualdata/Tesis/Basic/Villanueva_EM/enPDF/Cap1.pdf

Feedforward neural network. (s.f.). Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Feedforward_neural_network

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation.* Upper Saddle River: Prentice Hall.

Heaton Research, Inc. (2018). *Encog Machine Learning Framework*. Obtenido de https://www.heatonresearch.com/encog/

Jia, Y. (s.f.). *Caffe*. Obtenido de Deep learning framework : http://caffe.berkeleyvision.org/

Karn, U. (9 de Agosto de 2016). *A Quick Introduction to Neural Networks.* Obtenido de the data science blog: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

LeCun, Y., Corina, C., & J.C. Burges, C. (s.f.). *THE MNIST DATABASE of handwritten digits*. Obtenido de THE MNIST DATABASE of handwritten digits: http://yann.lecun.com/exdb/mnist/

LISA lab. (2008-2017). *Theano*. Obtenido de http://deeplearning.net/software/theano/

Malinova, A., Golev, A., Lliev, A., & Kyurkchiev, N. (04 de 08 de 2017). A FAMILY OF RECURRENCE GENERATING ACTIVATION FUNCTIONS BASED ON GUDERMANN FUNCTION. *International Journal of Engineering Researches and Management Studies*, 38-48. Obtenido de https://www.researchgate.net/publication/319158251_A_Family_of_Recurrence_Generating_Activation_Functions_Based_on_Gudermann_Function

National Institute of Standards and Technology. (2010-2018). *Digital Library of Mathematical Functions*. Obtenido de Gundermannian Function: https://dlmf.nist.gov/4.23#viii

Nielsen, M. (s.f.). *Neural Networks and Deep Learning*. Obtenido de Neural Networks and Deep Learning: http://neuralnetworksanddeeplearning.com/chap1.html

Olah, C. (27 de 08 de 2015). *Understanding LSTM Networks*. Obtenido de http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Sharma V., A. (30 de 03 de 2017). *Understanding Activation Functions in Neural Networks.* Obtenido de The Theory Of Everything: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

Sharma, S. (06 de 09 de 2017). *Activation Functions: Neural Networks*. Obtenido de Towards Data Science: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Skymind. (2018). *Deep Learning for Java*. Obtenido de https://deeplearning4j.org/

SourceForge. (s.f.). *Neuroph*. Obtenido de Java Neural Network Framework: http://neuroph.sourceforge.net/

Surmenok, P. (12 de 11 de 2017). *Estimating an Optimal Learning Rate For a Deep Neural Network.* Obtenido de https://towardsdatascience.com/estimating-optimal-learning-rate-for-a-deep-neural-network-ce32f2556ce0

Tchircoff, A. (04 de 08 de 2017). *The mostly complete chart of Neural Networks, explained.* Obtenido de https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

TensorFlow. (s.f.). Obtenido de An open source machine learning library for research and productionn: https://www.tensorflow.org/?hl=es