BIZKAIAKO INGENIARITZA ESKOLA

ESCUELA DE INGENIERÍA DE BILBAO

TÍTULO : IMPLEMENTACIÓN DE REDES NEURONALES EN PLATAFORMAS HARDWARE PARA SU APLICACIÓN EN INGENIERÍA ELÉCTRICA

MASTER EN INTEGRACIÓN DE LAS ENERGÍAS RENOVABLES EN EL SISTEMA ELÉCTRICO

AUTOR: XABIER MAESTRE BETOLAZA

TUTORES: JULEN GOMEZ-CORNEJO BARRENA, VICTOR VALVERDE SANTIAGO

CURSO: 2020/2021

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

ÍNDICE DE ABREVIACIONES

1. INTRODUCCIÓN

Las Redes Neuronales Artificiales (RNAs) son modelos computacionales que surgieron como un intento de conseguir formalizaciones matemáticas acerca de la estructura del cerebro. Imitan la estructura hardware del sistema nervioso, más exactamente, del cerebro humano. Se basa en el aprendizaje a través de la experiencia. [1]

Las RNAs están formadas por múltiples elementos procesadores, llamados neuronas. Las neuronas, organizadas en capas, están interconectadas entre ellas. El aprendizaje se consigue ajustando la fortaleza de las conexiones entre neuronas, para conseguir los resultados deseados.

La tecnología basada en RNAs se ha desarrollado rápidamente en las últimas décadas, y se ha aplicado a múltiples campos como en robótica, procesamiento de datos, clasificación y reconocimiento de patrones, sistemas de predicción, diagnóstico médico, y muchos más.

En el ámbito de la Ingeniería Eléctrica, se han utilizado para afrontar varios problemas, principalmente para la predicción de cargas, diagnosis de faltas, diseños en sistemas de control y estabilidad, sistemas de protección, análisis de seguridad de la red, control de potencia reactiva y voltaje y estudios de estabilidad transitoria.

Aunque la mayoría de las aplicaciones existentes de redes neuronales se desarrollan a menudo en forma de software, existen aplicaciones específicas en las que se requieren capacidades de procesamiento en paralelo mediante una implementación hardware que procese los datos en tiempo real, que sea capaz de entrenarse en tiempos razonables con grandes bases de datos, y que su consumo de energía sea eficiente. Las ventajas de utilizar redes neuronales en hardware especializado son la alta velocidad computacional que se consigue, una reducción en el coste de implementación de las redes, y una menor degradación de los componentes, por su diseño específico. [2]

En la actualidad, hay múltiples opciones de implementación en plataformas hardware de redes neuronales. En el presente trabajo se analizan las diferentes opciones de integración de RNAs, centrado principalmente en las plataformas Low Cost. También se recogen las pruebas de implementación realizadas en un Arduino y una Raspberry Pi.

2. ALCANCE Y OBJETIVOS

A continuación, se listan los objetivos del presente trabajo.

- Comprender el funcionamiento de las redes neuronales, y entender los diferentes algoritmos en los que basan su funcionamiento.

- Analizar el funcionamiento los principales tipos de redes neuronales que existen.

- Realizar un estudio sobre el tipo de redes neuronales que se utilizan en el campo de la Ingeniería Eléctrica, y sus aplicaciones.

- Analizar las diferentes alternativas de desarrollo e implementación de redes neuronales mediante software, y hablar de los requisitos hardware de los PC para el desarrollo de redes neuronales.

- Analizar y comparar las diferentes tecnologías que se han utilizado durante los años para implementar redes neuronales en hardware, y las nuevas plataformas hardware para la implementación software de redes neuronales.

- Implementar dos redes neuronales, una en Arduino y otra en una Raspberry Pi. Analizar el funcionamiento de las redes en estas plataformas.

- Realizar unas conclusiones y mencionar aspectos interesantes, guías futuras de desarrollo y mejoras sobre el trabajo, que puedan dar pie a trabajos futuros.

3. CONTEXTO

3.1. Fundamentos de las redes neuronales biológicas

Las RNA se crean con el objetivo de intentar imitar el proceso computacional y el razonamiento del cerebro humano, basándose en los modelos neuronales biológicos. [3]

En la Figura 1, se puede observar cuál es la unidad fundamental de las redes neuronales biológicas, la neurona. El sistema nervioso humano está formado por alrededor de mil millones de neuronas interconectadas entre sí, por más de mil billones de interconexiones. Es decir, el grado de interconexión en redes neuronales biológicas es enorme. Entre ellas, se transmiten información de unas a otras, en forma de sinapsis. La sinapsis es el proceso químico en el que una neurona "cargada" con una señal eléctrica transmite información a otras neuronas.

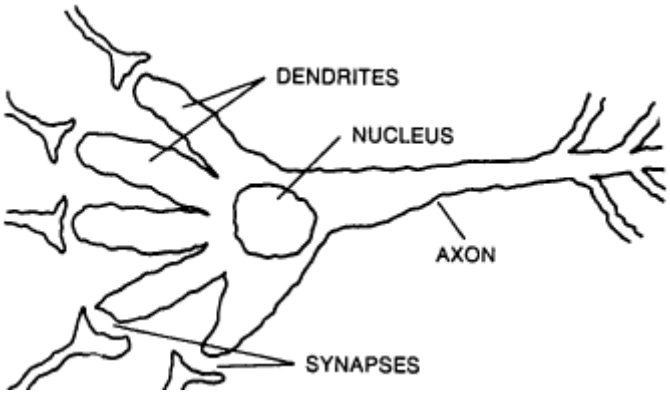

Figura 1 Esquema simplificado de una neurona. [4]

La célula consta, esencialmente de las siguientes partes:

- Dentritas, los elementos de entrada de información. A cada dentrita llegan sinapsis procedentes de otras neuronas.
- Núcleo o soma, el elemento de procesamiento.
- Axón, elemento de salida. Transmite las sinapsis a otras neuronas.

La conexión entre neuronas se realiza a través de contactos funcionales altamente especializados que se denominan sinapsis. Pueden ser de tipo químico o eléctrico.

Las características de las uniones entre neuronas no son fijas, cambian con el tiempo, con el objetivo de mejorar su funcionamiento. El aprendizaje de las neuronas se debe a la capacidad de variar estas características con el tiempo. Las conexiones entre neuronas, se

refuerzan o se debilitan, así consiguiendo que las neuronas se ajusten. De esta manera se consigue que grupos de neuronas se especialicen para ciertas tareas.

3.2. Fundamentos de las RNAs

3.2.1. Historia de las redes neuronales

Los primeros análisis acerca del comportamiento del cerebro y el proceso del pensamiento se deben a filósofos griegos como Aristóteles (384-422 a.C.) y Platón (427-347 a.C.). La investigación fue continuada por el filósofo, matemático y físico Descartes (1596-1650) y por los filósofos empiristas del siglo XVIII. [1]

En 1936, Alan Turing fue el primero en estudiar el cerebro desde el punto de vista computacional, aunque no fue hasta 1943 cuando el neurobiólogo Warren McCulloch, y el estadístico Walter Pitss, publicaron el artículo "A logical calculus of Ideas Imminent in Nervous Activity", donde presentaron los primeros modelos abstractos de neuronas artificiales. Así se constituyó la base del desarrollo en campos como son los Ordenadores Digitales (John Von Neuman), la Inteligencia Artificial (Marvin Minsky, sistemas expertos) y el funcionamiento del ojo (Frank Rosenblatt, red Perceptron).

Pocos años más tarde, en 1949 HEBB propuso una ley explicativa del aprendizaje neuronal conocida como la regla de Hebb que se convirtió en la antecesora de las modernas técnicas de entrenamiento de RNAs.

En 1956 se celebró la primera conferencia de Inteligencia Artificial, donde se reunieron los pioneros de Inteligencia Artificial de la época (Minsky, McCarthy, Rochester, Shanon). Esta conferencia marcó el punto de partida para el resto de investigadores del mundo, presentando las redes neuronales como una rama de investigación atractiva.

En 1957, Frank Rosenblatt publicó el mayor trabajo de investigación en computación neuronal realizado hasta esas fechas, donde desarrolló el elemento llamado Perceptron. Éste es un sistema clasificador de patrones, que identifica patrones geométricos y abstractos. El perceptron inicialmente se diseñó para reconocer ópticamente patrones mediante una rejilla de fotocélulas. Más adelante, Minsky y Papert encontraron limitaciones en el perceptron al resolver pequeñas tareas o problemas sencillos. En la década de los 60, se introdujo uno de los mayores cambios al perceptron de Rossenblat, creando sistemas multicapa, pudiendo así aprender y categorizar datos más complejos.

En 1959, Widrow desarrolló un elemento adaptativo lineal llamado Adaline (Adaptative Linear Neuron). Fue utilizada en múltiples aplicaciones, como reconocimiento de voz y caracteres, predicción del tiempo, control adaptativo y, principalmente, en el desarrollo de filtros adaptativos para eliminar ecos en las llamadas telefónicas.

A mediados de los años 60, Minsky y Papert (pertenecientes al Laboratorio de Investigación de Electrónica del MIT) publicaron un trabajo profundo de crítica al perceptron, transmitiendo a la comunidad científica que el perceptron y la computación neuronal no eran temas interesantes que estudiar. A partir de ese momento, las inversiones en la investigación de computación neuronal cayeron drásticamente. Hubo investigadores como James Anderson, Teuvo Kohonen y Stephen Grossberg que, a pesar del mal presagio que indicaron Minsky y Papert, continuaron con sus investigaciones.

En 1982, John Hopfield publicó los artículos "Hopfield Model" y "Crossbar Associative Network", e inventó el algoritmo de BackPropagation. De esta manera, se consiguió devolver el interés al campo de la computación neuronal, tras haber sufrido dos décadas de desinterés y casi total inactividad.

Actualmente, existen muchos grupos de investigación en todo el mundo realizando trabajos de investigación en el área de las RNAs. Este resurgimiento en los últimos años se ha producido debido al desarrollo teórico de nuevos modelos matemáticos y por el desarrollo de nuevas tecnologías. Las redes neuronales están siendo utilizadas en una gran variedad de aplicaciones comerciales desde hace varios años.

3.2.2. Computación tradicional vs computación neuronal

El modelo de computación neuronal difiere mucho del modelo de computación convencional en el que basan su comportamiento la mayor parte de procesadores y controladores del mercado. La computación neuronal, basada en el funcionamiento del cerebro biológico y las neuronas, tiene un funcionamiento y características diferentes. En la Tabla 1 se pueden observar una comparación entre los dos modelos de computación.

Tabla 1 Comparación entre la computación convencional y el cerebro humano. [1] [3]

	Computación neuronal	**Computación convencional**
Velocidad de proceso	Entre 10^{-3} y 10^{-2} s	Entre 10^{-8} y 10^{-9} s
Estilo de procesamiento	Paralelo	Secuencial (en serie)
Número de procesadores	Entre 10^{11} y 10^{14}	Pocos
Conexiones	10.000 por procesador	Pocas
Almacenamiento del conocimiento	Distribuido	En direcciones fijas (posiciones precisas)
Tolerancia a fallos	Amplia	Nula
Tipo de control del proceso	Auto organizado (democrático)	Centralizado (dictatorial)
Consumo de energía para ejecutar una operación/s	10^{-16} Julios	10^{-6} Julios

Las RNAs tienen una serie de ventajas y desventajas frente a la computación convencional. [5] Las ventajas de las RNAs son:

- Almacenamiento del conocimiento distribuido a lo largo de la red. A diferencia de la programación tradicional, se almacena en la red, no en una base de datos. La desaparición de alguna parte de la información no evita que la red deje de funcionar.

- Es capaz de trabajar con datos incompletos. Después de haber sido entrenada, la red es capaz de producir salidas aun introduciéndole información incompleta. El error cometido dependerá de la importancia de la información faltante.

- Tolerancia ante fallos. La destrucción de una o más células de una RNA no evita que se genere una salida.

- Capacidad de aprender. Aprenden de los eventos y toman decisiones en base a la experiencia obtenida en eventos similares.

- Capacidad de procesamiento en paralelo. Las RNAs tienen una potencia numérica tan grande que pueden realizar más de un trabajo a la vez.

Desventajas de las RNAs:

- Dependientes del hardware. Las redes neuronales requieren procesadores con capacidad de procesamiento en paralelo, de acuerdo con su estructura.

- Funcionamiento inexplicable de la red. Es uno de los mayores problemas de las RNAs. Cuando una red produce una solución, no da ninguna pista de por qué la ha escogido o cómo. Esto reduce la confianza en la red.

- Elección de la red neuronal adecuada. No hay ninguna regla específica para determinar el tipo y la estructura de las redes neuronales, aunque algunos tipos de redes neuronales destaquen para algunos usos. Las RNAs adecuada se diseñan gracias a la experiencia y mediante prueba y error.

- Dificultad para introducir los datos a la red. Antes de ser introducidos, los problemas y los datos relativos a ellos tienen que ser traducidos a valores numéricos.

- El resultado de la red neuronal no es óptimo, y es posible que en ocasiones la red dé resultados erróneos.

- Corrupción gradual. Las redes neuronales se ralentizan con el tiempo y sufren degradación relativa.

3.3. Funcionamiento de las RNAs

3.3.1. La neurona

Las RNAs tratan de emular el sistema nervioso biológico a través de procesadores artificiales. Para conseguir esto se utiliza un sistema compuesto por múltiples unidades de procesamiento simples, las neuronas. [1]

Las neuronas están altamente interconectadas, y son capaces de reproducir sus características de cálculo a partir de su auto organización. La función principal de las neuronas es recibir la información del exterior u otras neuronas, y procesarla de forma sencilla, proporcionando la salida al exterior u otras neuronas. En la Tabla 2 se realiza una analogía entre la neurona biológica y la neurona artificial.

Tabla 2 Analogía entre una neurona biológica y una neurona artificial. [3] [1]

Neurona biológica	Neurona artificial
Señales que llegan a la sinapsis	Entradas a la neurona, $x_i(t)$. Conjunto de valores que representan la información de entrada.
Carácter excitador o inhibidor de la sinapsis de entrada	Pesos sinápticos, $w_{ij}(t)$. Valor numérico que cuantifica la intensidad de la conexión.
Estímulo total de la neurona	Regla de propagación, $\sigma_i(t)$. Función que relaciona los pesos sinápticos y las entradas, proporcionando el potencial postsináptico $h_i(t)$.
Activación o no de la neurona	Función de activación, $F_i(h_i(t))$. Relaciona el estado de activación actual con el estado de activación anterior y el potencial postsináptico. El estado de activación, $a_i(t)$. Nivel de activación de cada neurona con un valor numérico.
Respuesta de la neurona	Función de transferencia o salida, $f_i(a_i(t))$. función que proporciona la salida final de la neurona.

Las reglas de propagación más comunes son las mostradas a continuación. [3]

Suma ponderada:

$$h_i(t) = \sum_j w_{ij} \cdot x_j \qquad \textbf{Ec. 1}$$

Distancia euclídea:

$$h_i(t) = \sqrt{\sum_j \left(x_j - w_{ij}\right)^2} \qquad \textbf{Ec. 2}$$

La mayoría de redes neuronales utilizan la función identidad, es decir, no consideran el argumento de activación anterior, el estado actual de la neurona no depende de su estado previo. La salida se acompaña del umbral de activación o bias, que se resta al potencial postsináptico. Representa el umbral de disparo de la neurona, haciendo que la respuesta de la red sea nula cuando el potencial postsináptico caiga por debajo del umbral de activación.

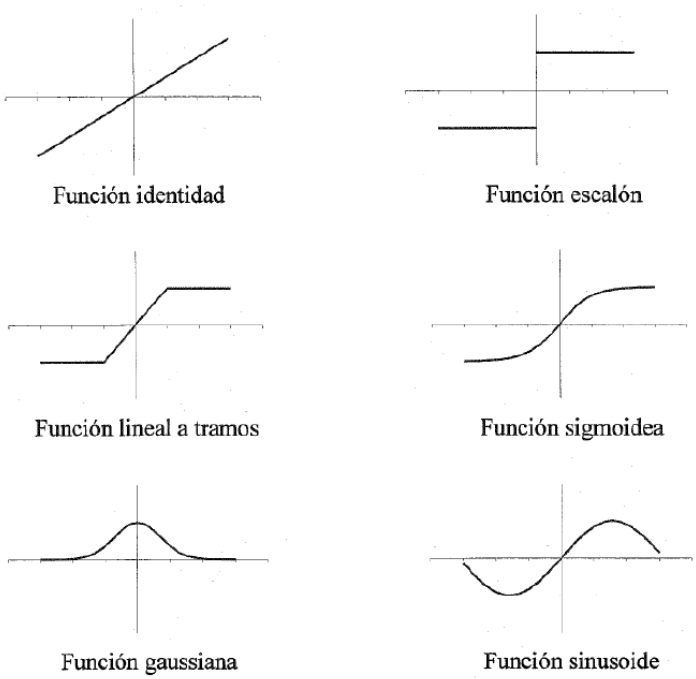

Figura 2 Funciones de transferencia más comunes. [3]

Las funciones de transferencia f_i más habituales aplicadas a RNAs son las que se muestran en la Figura 2, siendo la sigmoidea, la tangente hiperbólica y la lineal o identidad las más utilizadas. A continuación, se explican las características de las más habituales. [1]

- La función lineal devuelve directamente el valor de activación de la neurona. Este tipo de función se utiliza en redes de baja complejidad. Las redes que utilizan este tipo de función son, por ejemplo, las que siguen el modelo Adaline. [1]

- La función escalón presenta salidas binarias ({0,1}, {-1,1}). Si la activación de la neurona es inferior a un determinado umbral, la salida se asocia a una determinada salida, y si es inferior a un umbral, se asocia a otra salida determinada. Las aplicaciones para este tipo de neuronas son limitadas, se restringe para problemas binarios. Las redes que suelen utilizar este tipo de funciones son el perceptron simple [6], la red de Hopfield discreta [7] y la neurona clásica de McCulloch-Pitts [8].

- La función lineal a tramos es una variante progresiva de la función escalón. Se puede considerar como una función lineal saturada en sus extremos. Se establece un límite superior e inferior. Para valores inferiores al límite inferior, la salida se asocia a una determinada salida. Para valores superiores al límite superior, la salida se asocia a otra determinada salida. Para valores intermedios entre los dos límites, se aplica la función lineal.

- La función sigmoidea se define en un intervalo con límites superiores e inferiores. Las más destacadas son la función sigmoide y la tangente hiperbólica. Se caracterizan por ser monótonas, es decir, presentan una derivada siempre positiva,

o igual a cero en sus límites asintóticos. La ventaja de este tipo de funciones respecto a las anteriores es que, al estar la derivada definida en todo el intervalo, se pueden utilizar algoritmos de entrenamientos más avanzados.

- La función gaussiana adopta la forma de la campana de Gauss, y con el centro, radio y apuntamiento adaptables, se convierte en una opción versátil. Se suelen aplicar a redes complejas con más de 2 capas ocultas, que requieren de reglas de propagación basadas en el cálculo de distancias cuadráticas entre vectores de datos de entrada y pesos de la red, como la regla de la distancia euclídea.

- La función sinusoidal genera salidas continuas en el intervalo [-1,1]. Se suele usar en los casos que requieren periodicidad temporal.

En la Figura 3 se muestra el modelo de neurona que más se utiliza. Una neurona i recibe n señales de entrada x_j. Cada conexión de entrada está ponderada por un peso sináptico w_{ij}, que se relacionan con los pesos mediante la regla de propagación (en el caso de la Figura 3, la de la suma ponderada). A eso se le resta el umbral de activación Θ_i, se le aplica la función de transferencia f_i, proporcionando la salida de la neurona y_i.

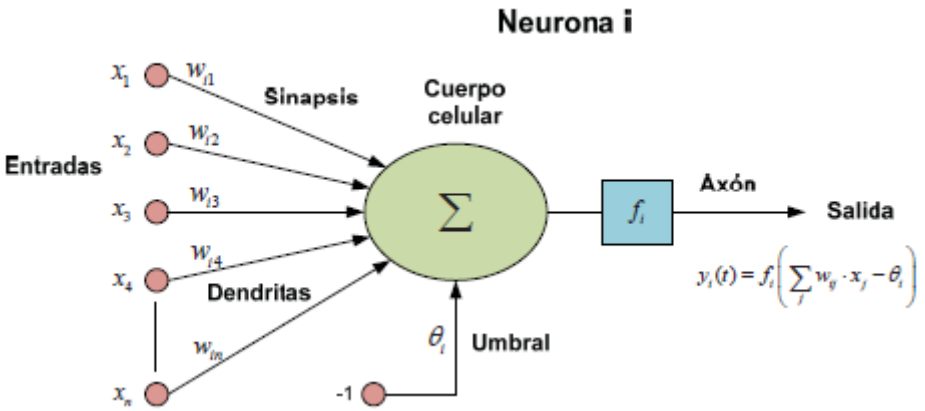

Figura 3 Modelo matemático de una neurona. [3]

En la Figura 4 se muestra esquemáticamente la secuencia de procesamiento de una neurona.

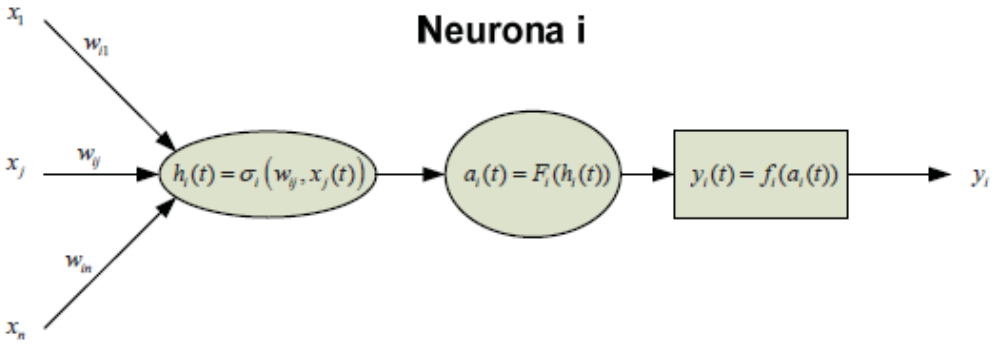

Figura 4 Secuencia de procesamiento de una neurona. [3]

3.3.2. Formación de redes

La capacidad de cálculo de la computación neuronal viene de conectar múltiples neuronas artificiales entre sí, construyendo así redes neuronales.

Las neuronas se agrupan principalmente en unidades estructurales llamadas capas. Las neuronas de las capas, se pueden conectar a su vez con otras capas, formando redes neuronales, como se aprecia en la Figura 5.

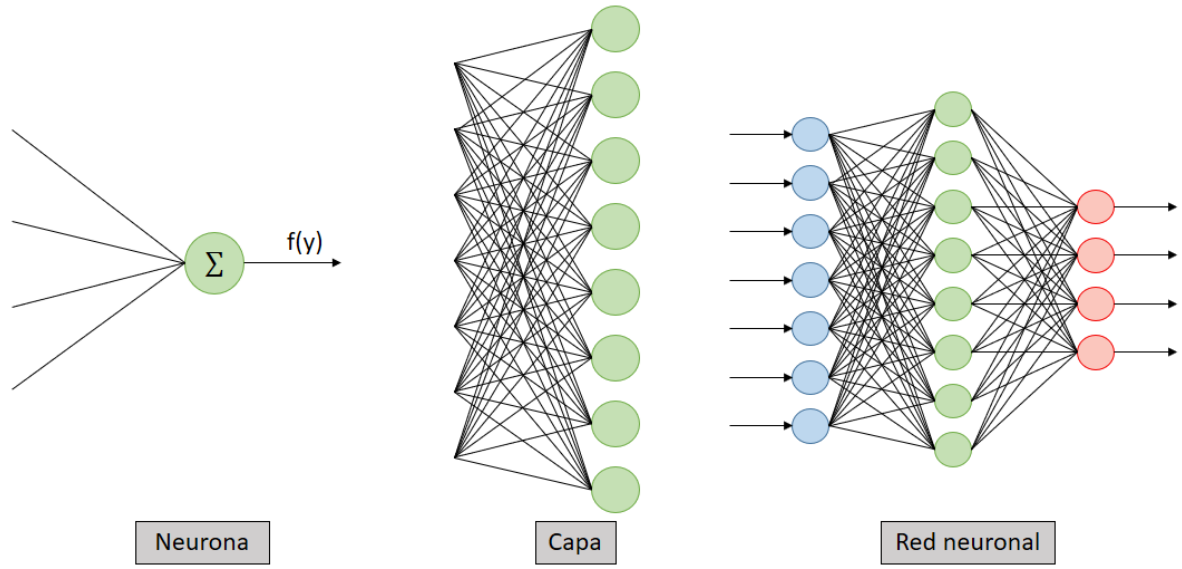

Figura 5 Neurona, capa y red neuronal.

En todas las redes neuronales, hay tres tipos de capas posibles:

- Capa de entrada, la encargada de recibir datos o señales procedentes del exterior.
- Capa oculta. Sus neuronas no tienen conexión directa al exterior. La función de esta capa es dar flexibilidad a la neurona, proporcionando más grados de libertad.
- Capa de salida. Sus neuronas proporcionan al exterior la respuesta de la red neuronal.

Las conexiones entre neuronas pueden ser:

- Intracapa, o conexiones laterales, tienen lugar entre neuronas de la misma capa.
- Intercapa, tienen lugar entre neuronas de diferentes capas. En algunos casos se dan conexiones realimentadas. La realimentación puede darse incluso dentro de una misma neurona.

Normalmente, cuanto más grandes y más complejas sean las redes, ofrecen mejores prestaciones en el cálculo. Las configuraciones de las redes existentes son muy variadas,

y cada tipo de red destaca en algunos aspectos sobre el resto, pero todas imitan la estructura en capas que presenta el cerebro. [9]

Las redes multicapa son superiores a las redes simples de una capa. En el caso de utilizar una función de activación no lineal, se mejora el funcionamiento general de la red, pudiendo simplificarla y reducir el número de neuronas.

En función del flujo de los datos en la red neuronal puede ser unidireccional o bidireccional. [3] [1]

- Redes unidireccionales, prealimentadas o de propagación hacia adelante (feedforward), cuando la información circula en un único sentido, de las neuronas de la capa de entrada hacia las neuronas de la capa de salida.

- Redes de propagación hacia atrás (feedback), cuando la información puede circular en cualquier dirección. Las salidas de las neuronas pueden servir de entradas a neuronas de capas anteriores. Se les llama redes recurrentes cuando se forman lazos cerrados. La realimentación también puede darse entre neuronas pertenecientes a la misma capa.

3.3.3. Modos de operación

A la hora de diseñar una red neuronal, es necesario distinguir dos modos de operación. Primero, la red pasa por una fase de entrenamiento, y cuando la red ya es capaz de generalizar respuestas, se pasa a la fase de recuerdo o ejecución, donde pasa a resolver nuevos problemas.

3.3.3.1 Entrenamiento o aprendizaje

Cuando se construye una red neuronal, se parte de un modelo de neurona, de una arquitectura de red, y los parámetros internos (pesos y bias) se establecen con valores iniciales aleatorios. A partir de ahí, es necesario que la red pase por la fase de entrenamiento antes de poder afrontar el problema objeto de estudio. [1]

Una de las principales características de las RNAs es la capacidad de aprendizaje que tienen. El objetivo del entrenamiento de las redes es conseguir que las salidas de la red se aproximen a las salidas objetivo. Para esto, el proceso de entrenamiento consiste en la aplicación secuencial de diferentes vectores de entrada, para ajustar los pesos sinápticos de las conexiones. Durante el entrenamiento, los pesos se ajustan gradualmente a valores que hacen que cada conjunto de datos de entrada produzca el conjunto de datos de salida deseado. [3]

Existen cuatro tipos predominantes de algoritmos de entrenamiento o procesos de ajuste de los pesos y los bias de las RNAs, en función de la existencia o no de un agente externo supervisor que controle el aprendizaje o no. Se pueden clasificar en supervisado, no supervisado, híbrido y reforzado. [9]

- Entrenamiento supervisado. Los algoritmos necesitan que los datos de entrada vayan emparejados con los datos de salida esperados. Se presenta un vector con los datos de entrada a la red, se calcula la salida de la red, y se compara con el

vector de datos de salida esperados. Se calcula el error cometido, y se utiliza para realimentar la red y realizar cambios en los pesos y bias mediante la utilización de un algoritmo que reduzca el error. En la Figura 6 se puede observar el diagrama de flujo genérico de los algoritmos de entrenamiento supervisados.

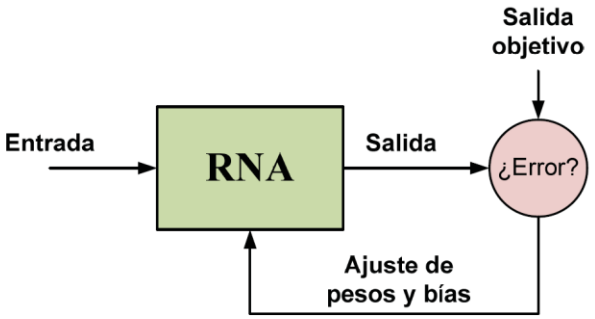

Figura 6 Diagrama de flujo de un algoritmo de entrenamiento supervisado. [3]

Las parejas de vectores del conjunto de entrenamiento se usan para el entrenamiento de forma cíclica. Se calcula el error y se ajusta los pesos por cada pareja entrada-salida hasta que el error es lo suficientemente pequeño para que sea aceptable.

Las redes neuronales con entrenamiento supervisado han recibido muchas críticas desde el punto de vista biológico, ya que no resulta lógico que en el cerebro exista un mecanismo que compare las salidas deseadas con salidas reales.

- Entrenamiento no supervisado o auto organizado. Este tipo de sistemas fueron desarrollados por Kohonen y otros investigadores. A diferencia de las supervisadas, no requieren de un vector de salidas deseadas, por lo que no se comparan salidas reales y esperadas. El algoritmo de entrenamiento modifica los pesos de la red de forma que produzca salidas consistentes. El proceso extrae propiedades estadísticas del conjunto de datos de entrenamiento, y permite agrupar patrones según su similitud.

- Entrenamiento híbrido. Combina el aprendizaje supervisado y no supervisado. Unas capas de la red se utilizan para el aprendizaje supervisado, y otras en cambio, se utilizan para el aprendizaje no supervisado.

- Aprendizaje reforzado. Es un aprendizaje con características del aprendizaje supervisado y no supervisado. No se le proporciona una salida esperada a la red, pero sí que se indica en mayor o menor medida el error global cometido.

Hoy en día existen muchos algoritmos de entrenamiento, y la mayoría de entrenamientos ha surgido de la evolución del modelo de aprendizaje no supervisado que propuso Hebb en 1949. Se caracteriza por incrementar el valor del peso de la conexión si las neuronas unidas son activadas. [9]

$$w_{ij}(n+1) = w_{ij}(n) + \alpha \cdot OUT_i \cdot OUT_j \quad \textbf{Ec. 3}$$

Es importante no confundir el nivel de error que se alcanza en la fase de entrenamiento, usando los datos de entrenamiento, con el error que comete la red ya entrenada, con entradas no utilizadas en el entrenamiento. Con este último se mide la capacidad de generalizar que tiene la red. Un indicador de que la red ha captado correctamente la relación entre los datos de entrada y salida, es tener un error de entrenamiento pequeño, con una buena capacidad de generalización. Para comprobar esto, es conveniente pasar por una fase de verificación y comprobar que la capacidad de generalizar de la red es la adecuada. [3]

3.3.3.2 Recuerdo o ejecución

Una vez que el sistema ha finalizado la fase de entrenamiento, la red se desconecta, haciendo que sus pesos y estructuras permanezcan fijos, quedando la red preparada para procesar nuevos datos, a partir del conocimiento extraído en el aprendizaje. [1]

3.4. Clasificación de las RNAs

La selección de una red (tipo de neurona, arquitectura de red y algoritmo de aprendizaje) se realiza en función de las características del problema a resolver. Es importante escoger la red más adecuada, para que el funcionamiento de la red sea óptimo. [1] [3]

En función de la topología y el tipo de aprendizaje que utilizan, se pueden destacar las siguientes redes neuronales más importantes (Tabla 3).

Tabla 3 Clasificación de redes neuronales en función de su tipo de aprendizaje y topología de red.

Tipo de Aprendizaje y Topología de red		Modelos de Redes Neuronales
Supervisado	Unidireccionales	Perceptron, Adalina, Madalina, MLP (Multi-Layer Perceptron), BackPropagation, GRNN (General Regression Neural Network), Máquina Boltzmann, Correlación en cascada
	Realimentados	BSB (Brain State in a Box), Mapa Fuzzy
No supervisado	Unidireccionales	LAM (Linear Associative Memory), OLAM (Optimal Linear Associative Memory), Mapas de Kohonen, Neocognitrón
	Realimentados	Adaptive Resonance Theory, Hopfield, BAM (Bi-directional Associative Memory)
Híbrido		RBF (Radial Basis Function), LVQ (Learning Vector Quantization), Contrapropagación
Reforzado		Aprendizaje reforzado

3.4.1. Perceptron simple

El perceptron simple es un modelo de red unidireccional, que está compuesto por dos capas de neuronas, una de entrada y otra de salida. En la Figura 7 se puede observar un perceptron simple de 8 neuronas en la capa de entrada y 4 neuronas en la capa de salida. El algoritmo de entrenamiento utilizado corrige el error cometido llevando actualizaciones discretas de los pesos. [3]

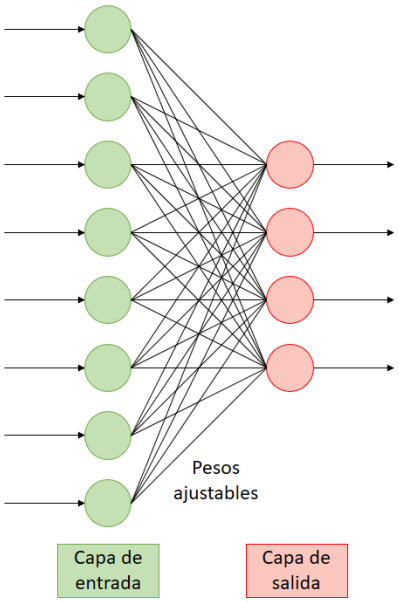

Figura 7 Perceptron simple.

Las neuronas de la capa de entrada no realizan ningún cómputo, simplemente envían la información a las neuronas de salida. La función de activación utilizada por las capas de salida es la función escalón. Una neurona tipo perceptron es un discriminador lineal, es decir, permite realizar tareas de clasificación. No es capaz de discriminar funciones no separables linealmente. Presenta limitaciones, que se pueden superar añadiendo más capas en la arquitectura de la red (perceptron multicapa).

3.4.2. Perceptron multicapa

El perceptron multicapa o red MLP es uno de los tipos de redes más utilizados, principalmente debido a su capacidad para adaptarse a prácticamente todo tipo de problemas. Mundialmente, se le conoce como un aproximador universal de funciones. [3]

Una red MLP es una red unidireccional que está formado por una capa de entrada, una cantidad variable de capas ocultas y una capa de salida. Las neuronas de cada capa están interconectadas con las de las capas adyacentes. La red MLP mostrada en la Figura 8 tiene una capa de entrada de 3 neuronas, dos capas ocultas de 5 y 4 neuronas respectivamente, y una capa de salida con 2 neuronas.

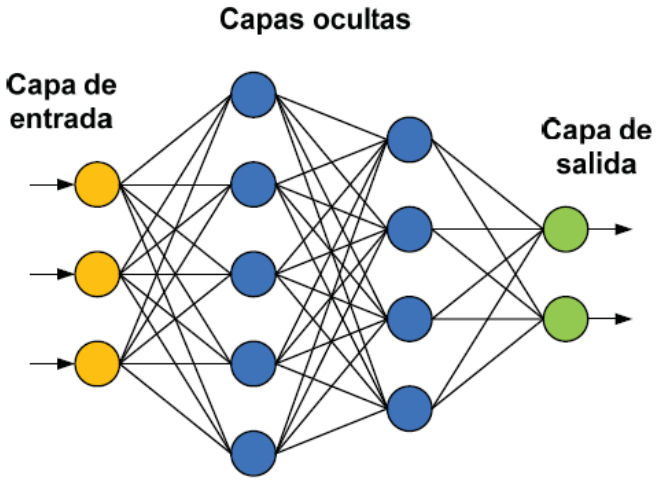

Figura 8 Perceptron multicapa. [3]

El algoritmo de aprendizaje utilizado para este tipo de red es el aprendizaje por retropropagación de errores o BackPropagation.

3.4.2.1 Retropropagación de errores

Las redes que se entrenan mediante la retropropagación del error o BackPropagation tienen un método de entrenamiento supervisado. A la red se le dan parejas de patrones, un patrón de entrada que está emparejado con un patrón de salida deseada. Por cada pareja de patrones, los pesos son ajustados de forma que se disminuye el error entre la salida deseada y la respuesta de la red.

Existen dos fases diferentes de entrenamiento, la primera conlleva una fase de propagación hacia adelante, y la segunda fase consiste en la propagación hacia atrás.

Propagación hacia delante

Primero, se inicializa la red, tomando unos valores iniciales de pesos y bias. Esta fase se inicia presentando los patrones de entrada a la red, y mediante las reglas de propagación explicadas previamente, se obtiene los patrones de salida.

Propagación hacia atrás

Una vez se ha completado la fase de propagación hacia adelante, se inicia la fase de propagación hacia atrás, retropropagación o fase de corrección. El objetivo de esta fase es minimizar el error cometido por la red en cada iteración.

El error que comete una red neuronal en función de sus pesos, crea un espacio de n dimensiones (n es el número de pesos de la conexión de la red). Mediante la regla del descenso del gradiente, se determina la dirección en la cual la función de error representa un mayor crecimiento. Se escoge la dirección negativa del gradiente, ya que el objetivo del proceso de aprendizaje es reducir el error.

La estructura y la nomenclatura de una red MLP genérica se puede ver en la Figura 9. Dado un conjunto de p entradas patrón, donde cada entrada se representa por un vector x de dimensión n_e, equivalente al número de neuronas de entrada de la red.

$$\bar{x}^{\mu} = (x_1^{\mu}, x_2^{\mu}, x_3^{\mu}, \dots, x_i^{\mu}, \dots., x_{n_e}^{\mu}) \qquad \text{siendo} \qquad \mu = 1, \dots, p \qquad \textbf{Ec. 4}$$

Cada entrada patrón se corresponde con un vector salida objetivo t de dimensión n_s, que es equivalente al número de neuronas de salida de la red.

$$\bar{t}^{\mu} = (t_1^{\mu}, t_2^{\mu}, t_3^{\mu}, \dots, t_k^{\mu}, \dots., t_{n_s}^{\mu}) \qquad \text{siendo} \qquad \mu = 1, \dots, p \qquad \textbf{Ec. 5}$$

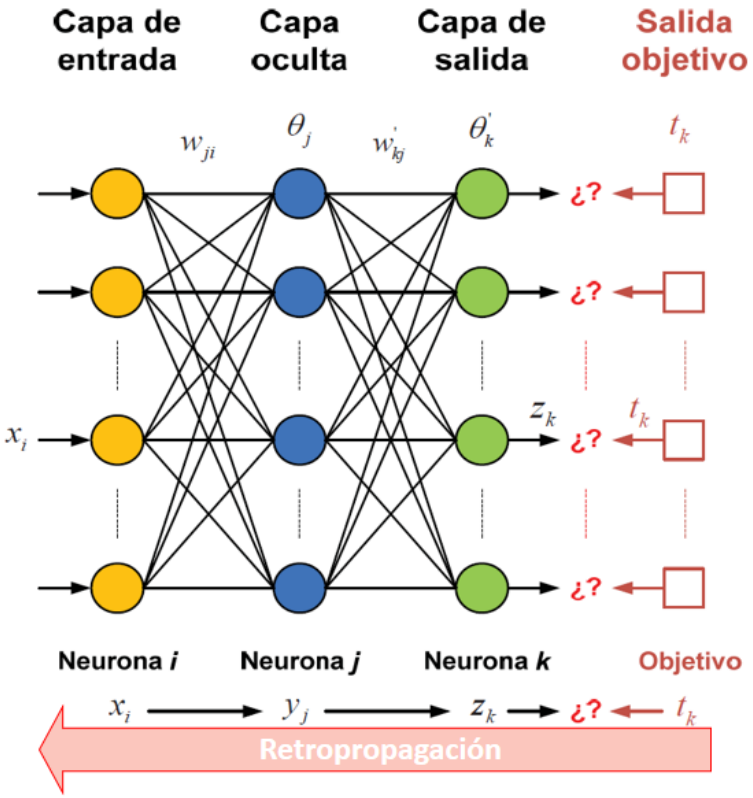

Figura 9 Nomenclatura y estructura de una red MLP de 3 capas. [3]

Siendo n_0 el número de neuronas de la capa oculta y f la función de transferencia de las neuronas de esa capa, la respuesta de la neurona k de la capa de salida de la red, para cada entrada patrón μ, se calcula:

$$z_k^{\mu} = \sum_{j=1}^{n_0} w'_{kj} \cdot y_j^{\mu} - \theta'_k = \sum_{j=1}^{n_0} w'_{kj} \cdot f\left(\sum_{i=1}^{n_e} w_{ji} \cdot x_i^{\mu} - \theta_j\right) - \theta'_k \qquad \textbf{Ec. 6}$$

Siendo y_j^{μ} la salida que presenta la neurona j de la capa oculta ante el patrón de entrada μ.

Para actualizar los pesos y bias, se calcula el error cometido por la red en cada iteración, para poder minimizarlo. La función a minimizar más común utilizada es el error cuadrático medio (RMS) del conjunto de patrones usados para el entrenamiento.

$$E(w_{ji}, \theta_j, w'_{kj}, \theta'_k) = \frac{1}{2 \cdot p \cdot n_s} \cdot \sum_{\mu=1}^{p} \sum_{k=1}^{n_s} (t_k^u - z_k^u)^2 \qquad \textbf{Ec. 7}$$

Aplicando la minimización del error según la regla del descenso de gradiente, se calculan las variaciones de los pesos de la capa oculta y de la capa de salida, que son proporcionales a la variación relativa del error con respecto a cada uno de estos pesos.

$$w'_{kj}(t+1) = w'_{kj}(t) + \Delta w'_{kj} \quad \text{siendo} \quad \Delta w'_{kj} = -\varepsilon \cdot \frac{\partial E}{\partial w'_{kj}} \qquad \textbf{Ec. 8}$$

$$w_{ji}(t+1) = w_{ji}(t) + \Delta w_{ji} \quad \text{siendo} \quad \Delta w_{ji} = -\varepsilon \cdot \frac{\partial E}{\partial w_{ji}} \qquad \textbf{Ec. 9}$$

Donde ε es la constante de proporcionalidad que se aplica al gradiente del error calculado.

El cálculo del error y la actualización de los bias sigue el mismo procedimiento que para los pesos. Empezando desde la capa de salida y continuando hacia atrás, se calculan las correcciones necesarias para los pesos y bias.

Los valores iniciales de pesos y bias influyen en la convergencia del algoritmo, por lo que es recomendable usar valores iniciales pequeños no nulos.

La utilidad de este algoritmo es tal, que durante los años se han ido desarrollando variantes de este algoritmo, mejorando su rendimiento. Entre estas variantes se encuentran métodos como el BackPropagation con momento, y el algoritmo de Levenberg-Marquardt.

3.4.3. RBF

Las redes de función de base radial (Radial Basis Function, RBF) son redes de prealimentada, que están formadas por tres capas: una capa de entrada, una capa oculta y otra de salida. En la Figura 10 se puede observar la configuración típica de una red RBF. [10]

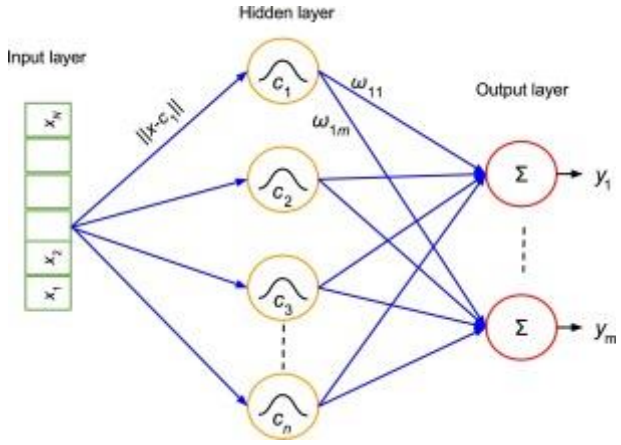

Figura 10 Red neuronal RBF.

La capa oculta consta de neuronas RBF con funciones de activación gaussianas (Figura 11). Las salidas de las neuronas RBF tienen respuestas significativas a las entradas solo en un rango determinado de valores, llamado campo receptivo.

Las neuronas de la capa de salida realizan una combinación lineal de las activaciones de las neuronas de las capas ocultas.

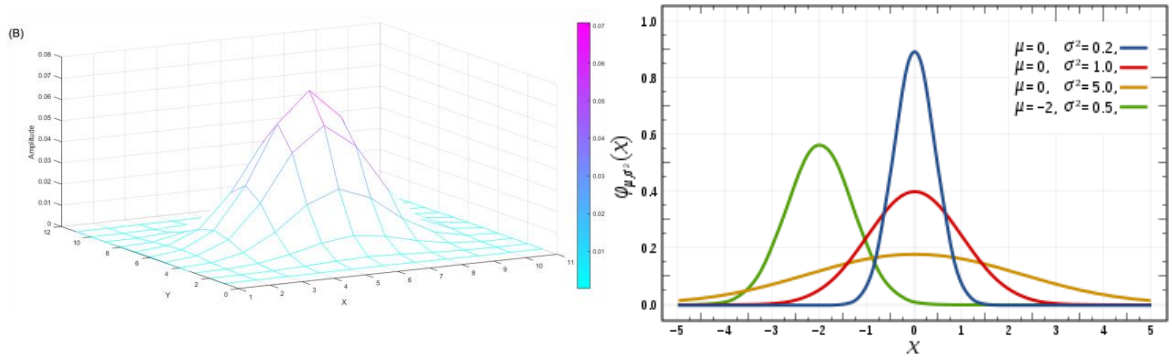

Figura 11 Representación gráfica de la función gaussiana.

Se tiene un conjunto de datos D que tiene N patrones de (x_p, y_p), dónde x_p es la entrada del conjunto de datos e y_p es la salida real. [11]

$$\phi_i(\| x - c_i \|) = e^{\left(-\frac{\|x-c_i\|^2}{2\cdot\sigma_j^2}\right)} \qquad \textbf{Ec. 10}$$

Donde $\| \cdots \|$ es la norma euclidiana, c_j y σ_j son el centro y la anchura, respectivamente, de la neurona j de la capa oculta.

Posteriormente, la salida del nodo k de la capa de salida de la neurona se calcula usando la Ec. 11.

$$y_k = \sum_{j=1}^{n} w_{jk} \cdot \phi_i(x) \qquad \text{Ec. 11}$$

La mayoría de redes RBF se entrenan en dos etapas. En la primera etapa se ajustan los centros y anchuras mediante algunos algoritmos no supervisados de clustering. En la segunda etapa, se ajustan los pesos de la capa oculta y la de salida mediante un entrenamiento supervisado, mediante el criterio de error RMS (Root Mean Squared Error).

Las redes RBF son un tipo de RNA utilizada principalmente para afrontar problemas de aproximación de funciones. Destacan respecto al resto de redes neuronales debido a su capacidad de aproximación y su velocidad de aprendizaje más rápida. Normalmente, una red RBF requiere menos tiempo para llegar al final del entrenamiento en comparación con una red MLP.

Entre sus aplicaciones se encuentran análisis de series temporales, procesamiento de imágenes, reconocimiento automático del habla, diagnósticos médicos, etc.

3.4.4. SOFM

La red de Kohonen, también conocida como Mapa auto organizado de características (Self-Organizing Feature Map, SOFM) es un tipo de red neuronal diseñada para aplicaciones en las que es importante mantener la topología entre los espacios de salida y entrada.

El entrenamiento de este tipo de redes es no supervisado, es decir, la red se auto-organiza. También son llamados mapas de características debido a que retienen las similitudes de los diferentes datos de entrenamiento, y se agrupan en función de las similitudes entre ellos. Esto se puede utilizar para visualizar grandes cantidades de datos de grandes dimensiones, y representar la relación entre ellos típicamente en mapas bidimensionales. [12]

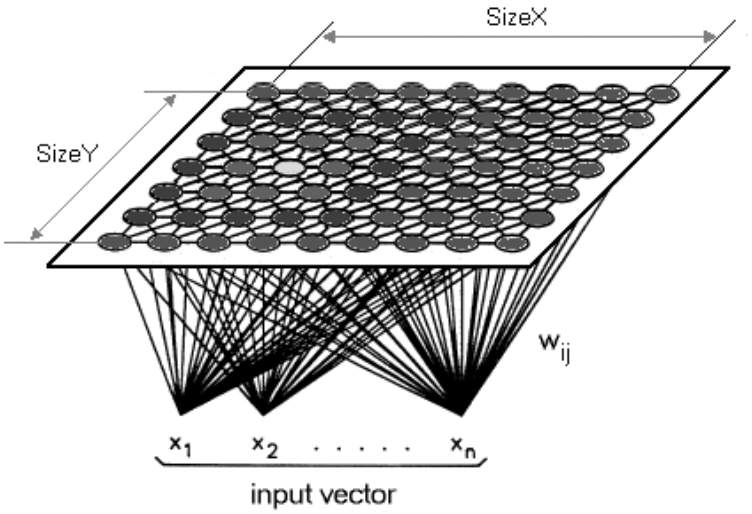

Figura 12 Mapa auto organizado SOFM. [12]

La estructura de las redes SOFM se puede apreciar en la Figura 12. Es una red de tipo unidireccional. La red se organiza en dos capas, siendo la inferior la capa de entrada y la superior la capa de salida. La capa de salida consiste en una matriz de neuronas de dos dimensiones. [13]

Todos los nodos de esta "cuadrícula" o capa están conectados directamente al vector de entrada, pero no entre sí, lo que significa que los nodos no conocen los valores de sus vecinos y solo actualizan el peso de sus conexiones en función de las entradas dadas. La cuadrícula en sí misma, es el mapa que se organiza en cada iteración en función de la entrada de los datos de entrada.

En vez de utilizar un aprendizaje mediante la corrección de errores, las redes SOFM utilizan un aprendizaje competitivo para ajustar sus pesos. Esto significa que solo se activa un único nodo en cada iteración, en el cual las características de un instante del vector de entrada se presentan a la red, mientras todos los nodos compiten por poder responder a la entrada.

El nodo elegido o la mejor unidad de coincidencia (Best Matching Unit, BMU) se selecciona de acuerdo con la similitud entre los valores de entrada actuales y todos los nodos de la cuadrícula. Se elige el nodo con la menor diferencia euclidiana al vector de entrada posible, junto con los nodos colindantes al nodo ganador dentro de un radio.

En la Figura 13 se puede ver un modelo de red SOFM, donde el BMU está marcado en amarillo, y las neuronas colindantes se marcan en rosa y morado.

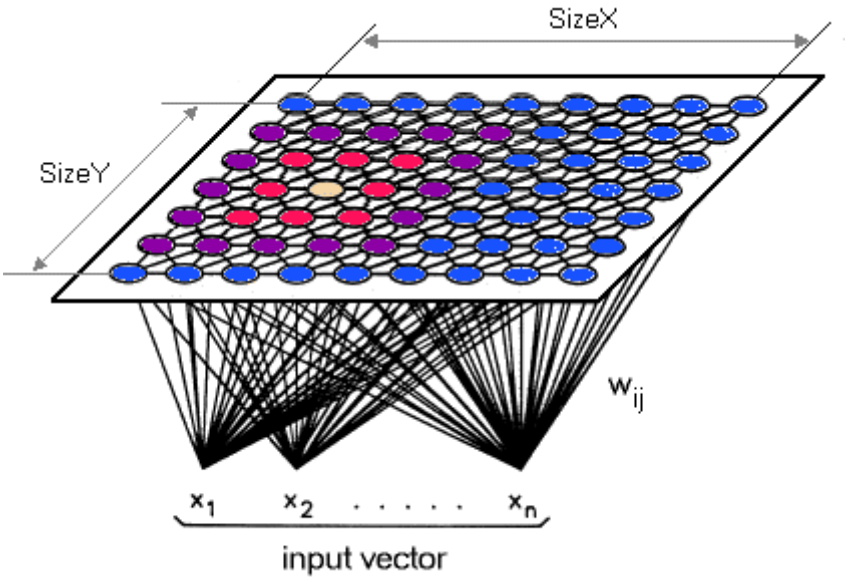

Figura 13 Red SOFM tras la selección del BMU y sus neuronas colindantes. [12]

Se actualiza el vector de pesos de los nodos colindantes del BMU (rosas), añadiendo una fracción de la diferencia entre el vector de entrada x(t) y el peso w(t) de la neurona.

Según va pasando por todos los nodos de la cuadrícula, toda la red coincide eventualmente con el conjunto de datos de entrada, con los nodos similares agrupados en un área, y los que son diferentes separados.

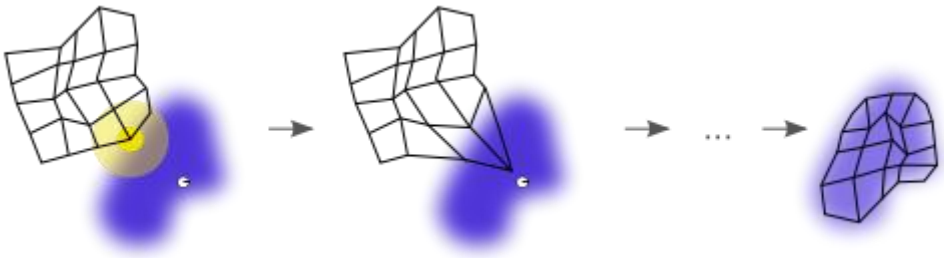

Figura 14 Proceso de entrenamiento de una red SOFM. [14]

En la Figura 14 se muestra el entrenamiento de un mapa auto organizado. La zona azul es la distribución de los datos de entrenamiento, y la cuadrícula blanca es el ejemplo de mapa auto distribuido. En amarillo, se muestra la primera BMU, que se desplaza junto con sus nodos colindantes. En la siguiente iteración, se puede ver cómo la red se va ajustando al conjunto de datos de entrada, hasta que finalmente se ajusta por completo.

3.4.5. RNN

Las redes neuronales recurrentes (Recurrent Neural Network, RNN) son un tipo de redes neuronales que permiten que las salidas anteriores se utilicen como entradas mientras tienen estados ocultos. [15]

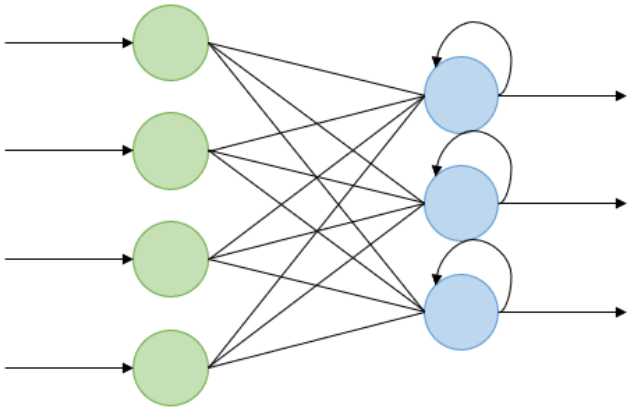

Figura 15 Red neuronal recurrente RNN.

Son un tipo de redes neuronales que se adaptan naturalmente al procesamiento de datos de series temporales y otros datos de carácter secuencial. Las RNN consisten en una variación de las redes neuronales prealimentadas, ya que permiten el procesamiento de secuencias de datos de longitud variable (teóricamente infinita). Las redes RNN más populares son las redes de memoria a corto plazo (Long Short-Term Memory, LSTM) y las unidades recurrentes controladas (Gatered Recurrent Units, GRUs).

Las redes prealimentadas parten tradicionalmente de entradas y salidas de dimensiones preestablecidas. Las RNN, en cambio, operan naturalmente con secuencias de entrada de longitud variable.

Las RNN producen resultados predictivos con datos secuenciales que otros algoritmos no pueden conseguir.

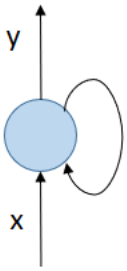

Figura 16 Neurona recurrente.

En la Figura 16 se representa una RNN de una única neurona que recibe una entrada, produce una salida, y esa salida se envía a sí misma como entrada para el siguiente instante. De la misma forma, una capa de neuronas recurrentes (Figura 17) se puede implementar de forma que, en cada instante de tiempo, cada neurona recibe dos entradas la correspondiente de la capa anterior, y a su vez la salida del instante anterior de la misma capa. [16]

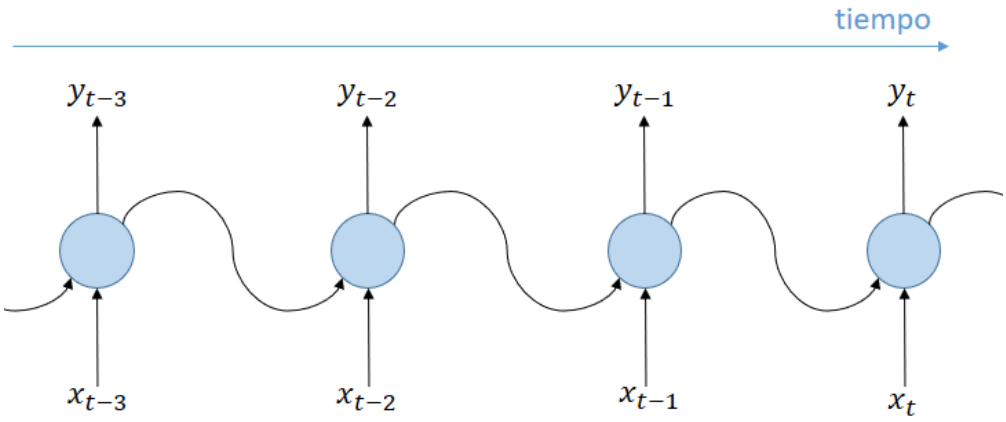

Figura 17 Representación de diferentes instancias de la RNN de una única neurona.

A diferencia de las redes prealimentadas tradicionales, las redes neuronales recurrentes pueden tener una longitud variable de datos entrada y de salida. En función de esto, hay varios tipos de redes neuronales recurrentes, como se muestra en la Figura 18. Con este tipo de redes se pueden usar para generación de música, clasificación de sentimientos y traducción artificial.

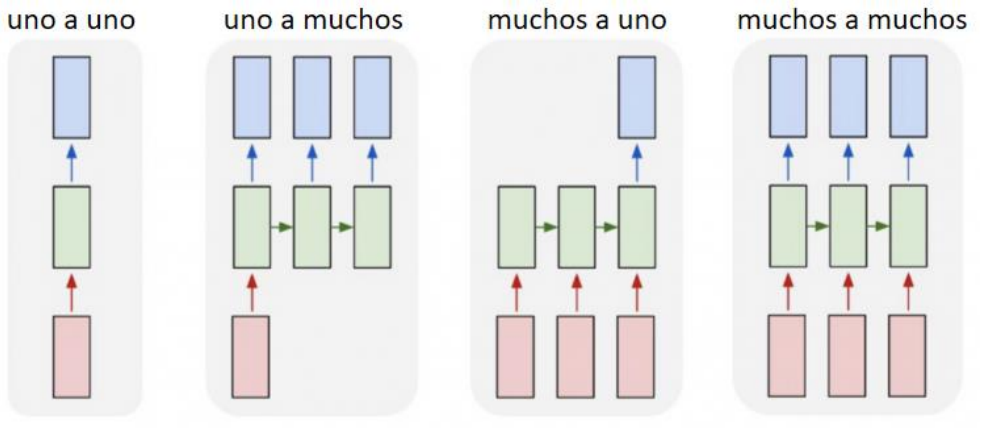

Figura 18 Tipos de RNNs. [17]

Existen dos fenómenos que afectan negativamente al funcionamiento del descenso del gradiente de las RNN:

- Gradientes Explosivos (Exploding Gradients): el algoritmo asigna un gradiente excesivamente alto a los pesos sin razón, generando un problema en el entrenamiento de la red. El problema se puede resolver truncando o reduciendo los gradientes.

- Gradientes Desaparecidos (Vanishing Gradients): cuando los valores de un gradiente son demasiado pequeños, el modelo puede dejar de aprender o requerir una cantidad de tiempo demasiado alta. El problema se resuelve mediante el concepto de puertas (Gate Units).

3.4.6. LSTM

Las redes de memoria a corto plazo (Long Short-Term Memory, LSTM) son una ampliación de las redes neuronales recurrentes, a las cuales se le extiende la memoria para aprender de experiencias importantes que hayan pasado hace mucho tiempo. Esto se debe a que las LSTM contienen su información en la memoria, donde una neurona puede leer, escribir y borrar información de ella en función de la importancia que se le asigna a esa información. Mediante ese proceso de escribir y borrar la memoria, la red aprende con el tiempo qué información es importante o no. [17]

En una neurona LSTM (Figura 19) hay tres puertas: una puerta de entrada (input gate), una puerta de olvidar (forget gate) y una puerta de salida (output gate).

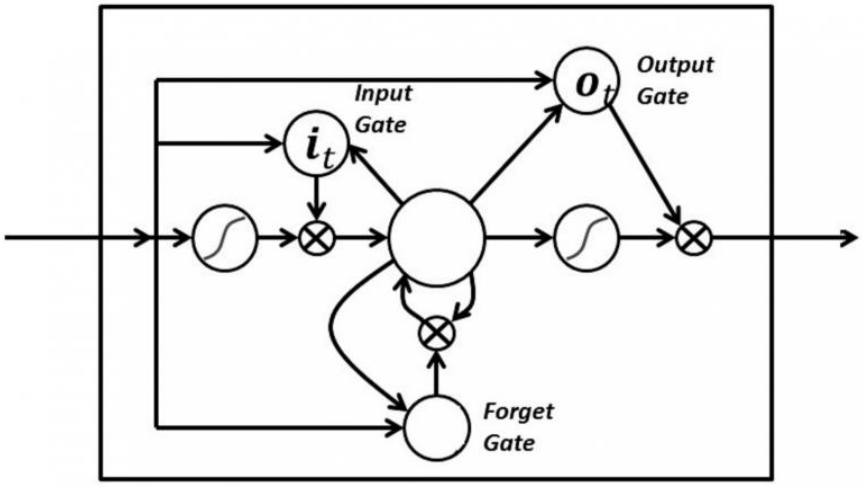

Figura 19 Neurona LSTM.

Mediante estas puertas se determina si se permite o no una nueva entrada, se elimina la información porque no es importante o se deja que afecte a la salida en el paso de tiempo actual. [16]

3.4.7. Red de Hopfield

La red de Hopfield (Hopfield Network) es un tipo de red neuronal recurrente, que consiste en una capa de neuronas recurrentes (Figura 16) totalmente interconectadas entre ellas. Este tipo de redes se utilizan cono sistemas de memoria asociativa con unidades binarias. Se dice que son binarias debido a que utilizan la función de transferencia tipo escalón. Es decir, solo puede tener dos valores posibles, 1 ó -1, o bien 1 ó 0. [7]

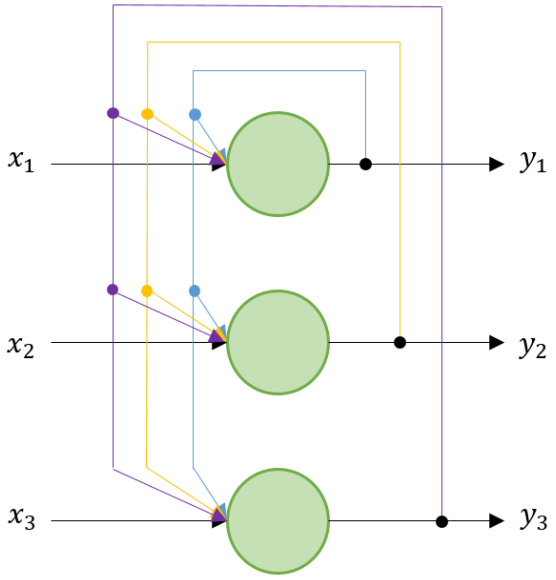

Figura 20 Red de Hopfield.

En la Figura 20 se puede ver una red de Hopfield de 4 neuronas. La salida de cada neurona se realimenta al resto de neuronas de la capa para la siguiente iteración, tal y como se ha comentado con las RNN. [18]

3.4.8. ConvNet

Las redes neuronales convolucionales (Convolutional Neural Network, CNN/ConvNet) son un tipo de redes neuronales que simula el funcionamiento de la corteza visual primaria de un cerebro. Es una variación del perceptron multicapa, donde la entrada es una matriz bidimensional en vez de un vector. Gracias a esto, es muy útil para tareas de visión artificial, como clasificación y segmentación de imágenes. [19]

En la Figura 21 se muestra una ConvNet que ha sido entrenada para el reconocimiento de dígitos. Como se puede ver, las ConvNet consisten en muchas capas de kernels o filtros convolucionales. Es capaz de identificar patrones espaciales y temporales en una imagen mediante la aplicación de filtros.

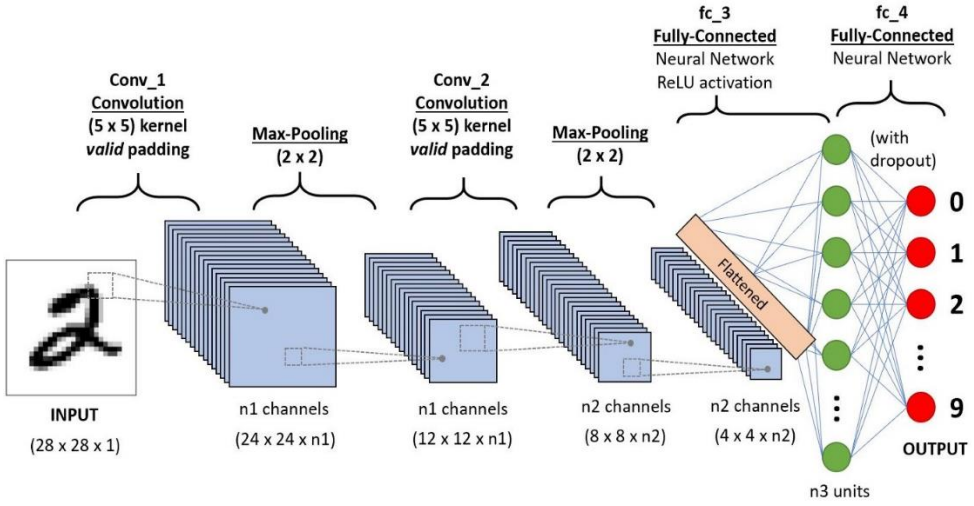

Figura 21 ConvNet para el reconocimiento de dígitos. [19]

En la Figura 22 se puede ver otro ejemplo de red ConvNet para el reconocimiento facial. Se puede ver cómo, mediante la combinación de diferentes filtros, es capaz de distinguir patrones cada vez más complejos, hasta ser capaz de diferenciar los rostros de diferentes personas.

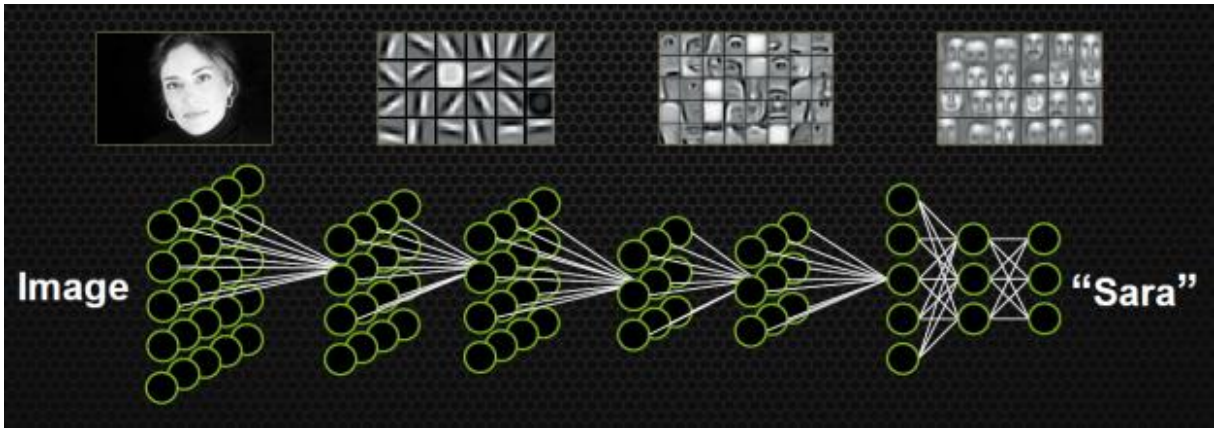

Figura 22 Representación esquemática de una ConvNet para el reconocimiento facial. [20]

3.4.8.1 Capas convolucionales

En la Figura 23 se puede observar una imagen RGB que se ha desglosado en los tres canales de colores rojo, verde y azul. En este caso, se tiene una imagen de 4 pixeles de ancho y de alto, y 3 canales de colores, siendo necesarias un total de 48 neuronas de entrada. En el caso de que se quisiera analizar una imagen de calidad estándar, por ejemplo, de 5.152 × 3.432 píxeles. En total, harían falta más de 53 millones de neuronas de entrada. El rol de las redes ConvNet es reducir las imágenes a una forma más sencilla para procesar, sin perder las características necesarias para realizar una buena predicción.

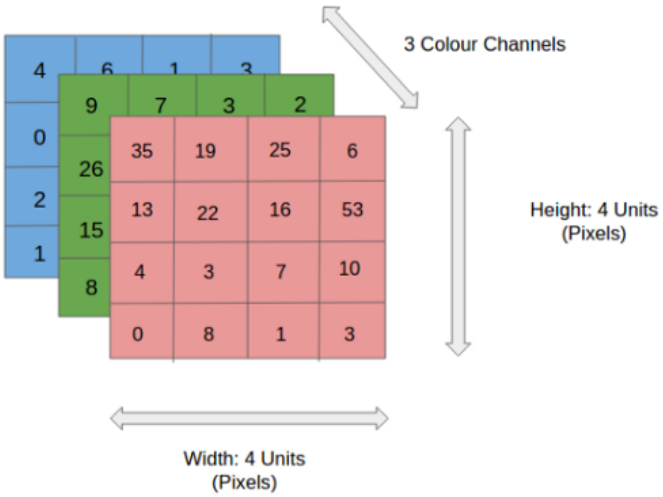

Figura 23 Imagen de entrada separada en capas RGB.

En la Figura 24 se muestra el proceso de aplicar un filtro a una imagen. La imagen de entrada (5x5x1) aparece representada en verde. El filtro o kernel es elemento que lleva a cabo la operación convolucional, viene representado en color amarillo. La característica convolucionada (convolved feature) está representado en la tabla de la derecha roja. Es una matríz de 3x3x1.

$$Kernel = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad \text{Ec. 12}$$

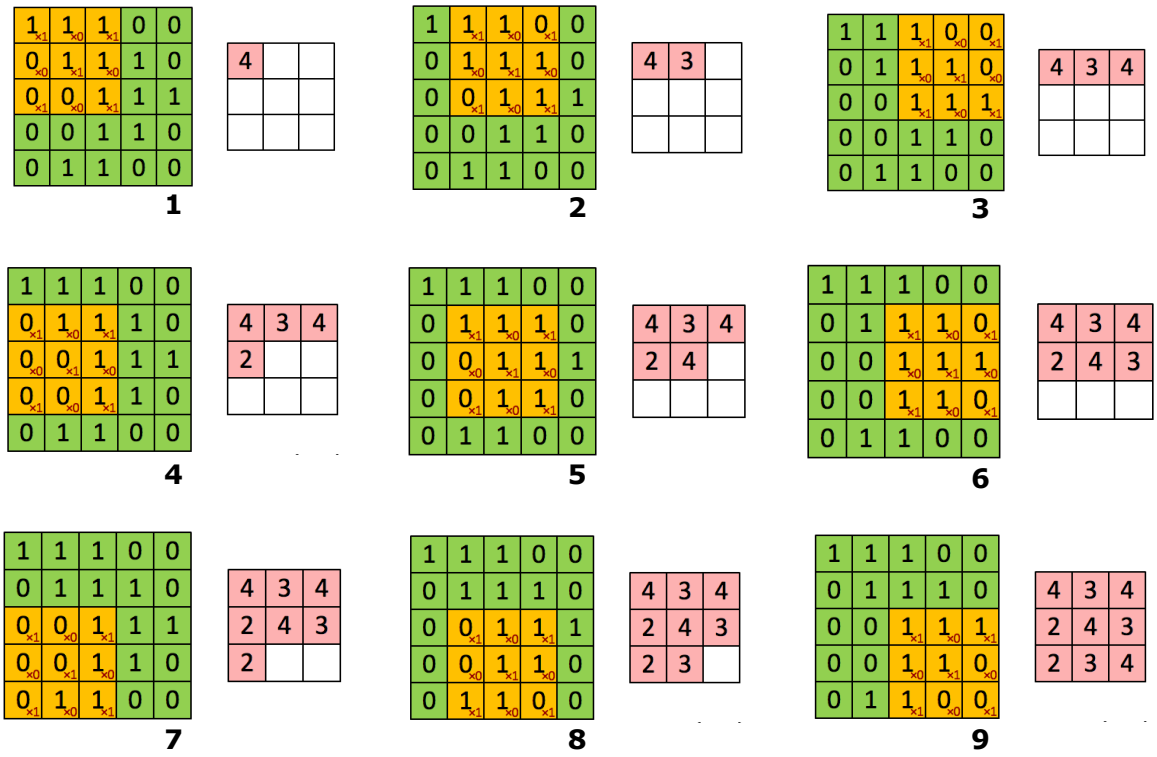

Figura 24 Aplicación de un filtro.

El kernel se mueve hacia la derecha con un determinado paso, y al llegar al lado derecho, se mueve con el mismo paso, a la siguiente línea, comenzando desde el márgen izquierdo. El proceso continua hasta que se ha completado toda la imagen. En este caso, el paso es 1, por lo que cambia 9 veces.

En los casos en los que se disponen más de un canal (RGB, escala de grises, HSV, CMYK, etc.), se suman los resultados de cada multiplicación de las diferentes capas, junto con el bias. De esta forma, la salida de la red solo tiene una profundidad de un canal.

El objetivo principal de estas operaciones de convolución es extraer las características de alto nivel de las imágenes. Las redes ConvNet tienen muchas capas convolucionales, siendo las primeras las encargadas de capturar las características de bajo nivel como colores, bordes, orientación de los gradientes, etc. En la Figura 25 se muestra un ejemplo de los kernels que se aplican a las figuras. Añadiendo más capas, se consigue que la red se adapte para detectar características de alto nivel, obteniendo una red que conoce de una forma más general de las imágenes de entrenamiento, similar a como lo haría un cerebro humano.

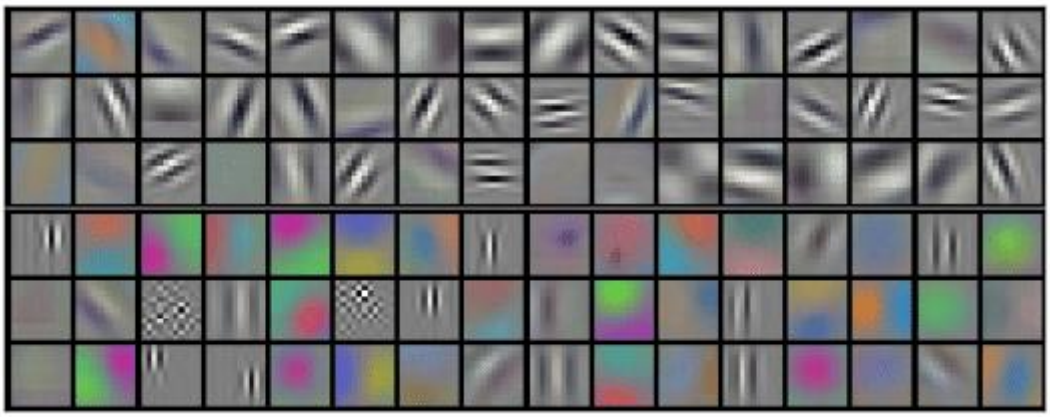

Figura 25 Ejemplo de kernels de una ConvNet. [21]

Hay dos tipos de resultados de operación:

- Valid Padding. La característica convolucionada de salida se reduce en tamaño respecto a la entrada recibida.

- Same Padding. La dimensión de la característica convolucionada de salida aumenta o se mantiene igual que la de la entrada recibida.

3.4.8.2 Capas agrupadoras

Las capas agrupadoras (Pooling Layer) tienen una función similar que las capas convolucionales, reducir la dimensión de las características convolucionadas. Esto se hace para reducir la potencia de computación necesaria para procesar los datos.

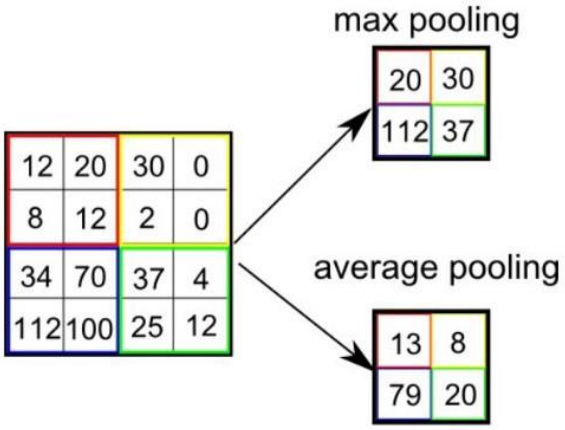

Figura 26 Max pooling y average pooling.

Hay dos formas de realizar agrupaciones. En la Figura 26 se puede observar un ejemplo para cada tipo de agrupación.

- Max Pooling. Devuelve el máximo valor del trozo de imagen cubierto por el kernel. También actúa como un supresor de ruido junto con la reducción de la dimensión. El funcionamiento del max pooling es superior al average pooling.

- Average Pooling. Devuelve la media del trozo de imagen cubierto por el kernel. La única supresión del ruido se realiza mediante la reducción de la dimensión.

Las capas convolucionales y las capas de agrupaciones, en parejas, constituyen la base de las redes convolucionales.

3.4.8.3 Capa de unidades lineales rectificadas

Las unidades lineales rectificadas (Rectified Linear Unit, ReLU Layer) consiste de una capa de neuronas que aplican una función de activación definida como:

$$f(x) = \max(0, x) \qquad \textbf{Ec. 13}$$

Elimina los valores negativos del mapa de activación, haciéndolos nulos. [22]

3.4.8.4 Capa de clasificación

Añadiendo al final una capa o más de neuronas totalmente interconectadas entre ellas, como las que se encuentran en un perceptron multicapa, se le añade a la red la capacidad de aprender combinaciones no lineales.

Para poder introducir los datos de entrada al perceptron multicapa, es necesario "aplanar" la imagen a un vector columna. Después se pre propagan las entradas y se retropropagan los errores. A través de la técnica de clasificación Softmax, se pueden distinguir características dominantes y algunas de bajo nivel en las imágenes.

4. ESTADO DEL ARTE

Las redes neuronales son consideradas una alternativa viable para afrontar problemas complejos o poco definidos. Son capaces de aprender de ejemplos, pueden manejar datos incompletos o con ruido, pueden solucionar problemas no lineales, y una vez entrenadas, pueden realizar predicciones y generalizaciones a una velocidad muy alta. [23]

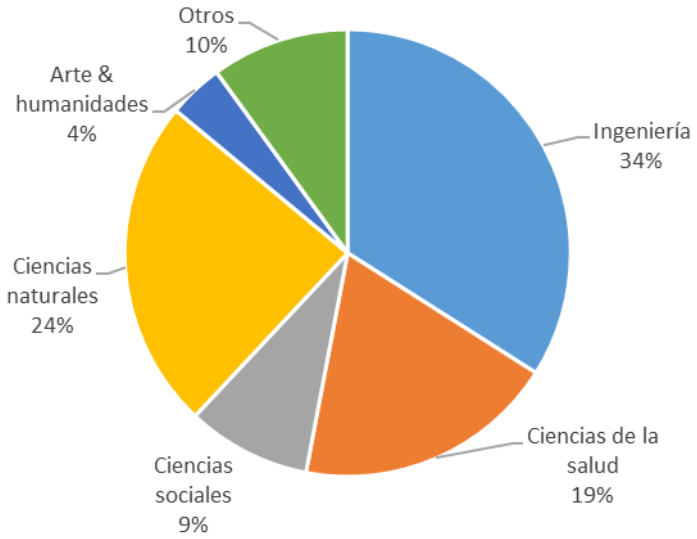

Figura 27 Distribución del uso de RNAs en diferentes áreas, 2013. [24]

Gracias a sus características, se han utilizado ampliamente en diversos campos, como se puede observar en la Figura 27: control, robótica, reconocimiento de patrones, predicciones, medicina, producción, optimización, y en aplicaciones de ciencias sociales y psicológicas, y también en el ámbito de la ingeniería eléctrica y los sistemas eléctricos de potencia.

4.1. Aplicación de las redes neuronales en ingeniería eléctrica

Las redes neuronales se han ido incorporando progresivamente en Ingeniería Eléctrica durante los años, a medida que se desarrollaba la tecnología para implementar redes mayores y con mejores funcionamientos. En la Figura 28 se pueden ver diferentes aplicaciones de las redes neuronales y su importancia dentro del sector de ingeniería eléctrica.

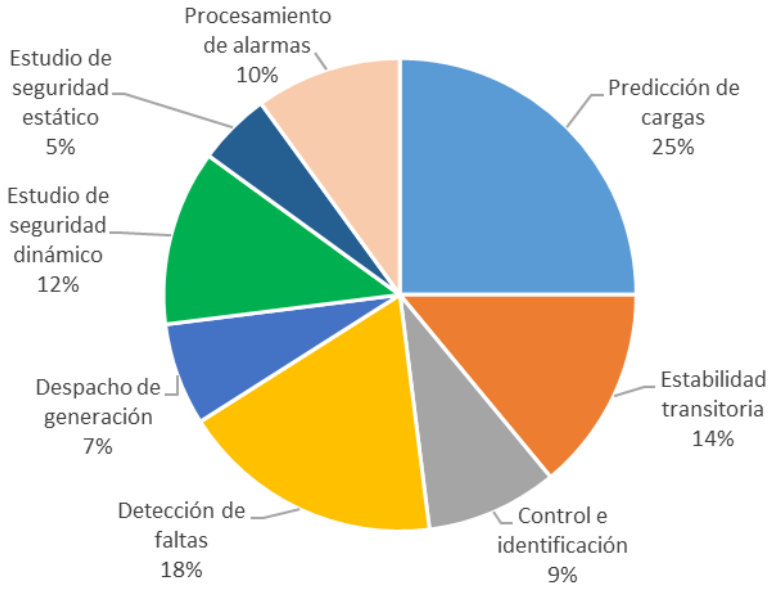

Figura 28 Aplicación de las redes neuronales en ingeniería eléctrica, 2005. [25]

4.1.1. Predicción de cargas

La habilidad de predecir los consumos en el sistema de forma precisa es probablemente el factor económico más importante en la operación eficiente del sistema eléctrico. [26]

El consumo de electricidad varía mucho para cada hora, día, semana, mes o año. Esto ocurre debido a muchos factores socioculturales, como las vacaciones, diferentes festividades o eventos o la climatología.

Es un problema adecuado para la aplicación de redes neuronales, debido a la alta disponibilidad de datos históricos para la fase de entrenamiento de las redes.

- Predicción de cargas a corto plazo, comúnmente de un periodo de 24 horas. No es un intervalo de tiempo fijo, en función de la situación se puede alargar hasta periodos de 1 semana. La información obtenida mediante estas predicciones ayuda a gestionar desvíos de energía. También se usa para coordinar la generación de diferentes generadores (Unit Commitment), y para planificar y operar la reserva.

- Predicción de cargas a medio plazo, con una duración de un mes a varios años. La información obtenida se usa para el cálculo del coste de los combustibles y las tarifas eléctricas. También se usa para planificar la operación y el mantenimiento de la red.

- Predicción de cargas a largo plazo, con una duración de 7 a 18 años. Esencialmente, se usan para determinar el tamaño y enfoque de futuras expansiones de la red, y reducciones de costes fijos y variables.

Las redes neuronales se desarrollan sobre todo para la predicción de cargas a corto plazo. Principalmente, los tipos de redes neuronales más utilizados son las SOFM y MLP con un algoritmo de BackPropagation. [27]

También destaca el uso para predicciones a largo plazo, con otros métodos de BackPropagation como RBF y RNN, aunque la más eficaz para este tipo de predicciones es RBF.

4.1.2. Despacho de generación

El despacho de generación (Economic Dispatch) de los generadores siempre ha sido un factor económico clave en la operación de los sistemas eléctricos de potencia. [25] Es un proceso computacional en el que la generación total requerida se distribuye entre las unidades de generación, minimizando los costes. Para cada escenario planteado, se determina la potencia a generar para cada planta. Se usa en sistemas de gestión de energía y control en tiempo real, principalmente para coordinar el coste de la producción de todas las plantas que operan en el sistema eléctrico.

El principal objetivo consiste en minimizar los costes, dependiendo de la demanda y de algunas limitaciones como la localización de las cargas, la localización de la generación y las características de la red.

Se han desarrollado muchas herramientas para realizar estos cálculos, como el método de relajación Lagrangiana, métodos de programación lineal, programación dinámica, programación cuadrática de Beale's, y muchos otros. También se han utilizado redes neuronales para realizar el despacho de generación. [25]

La programación de la generación de las centrales térmicas e hidráulicas es un problema similar a una combinación de la predicción de cargas y el despacho de generación. La programación de la generación de este tipo de centrales se tiene que enfocar teniendo en cuenta los costes del combustible, costes de operar al ralentí, y costes de arranque. También hay que tener en cuenta las limitaciones para transportar energía que tiene la red. La gestión de las centrales hidráulicas hay que hacerla teniendo en cuenta que, aunque tengan unos costes operacionales muy pequeños, está sujeta a la cantidad de agua disponible para el turbinado. Esto aumenta la complejidad de programación de la generación. [28]

Principalmente, se han utilizado el tipo de redes neuronales de Hopfield y variaciones de ésta, debido a su memoria donde se almacenan patrones e información sobre eventos pasados, que le ayudan a tomar mejores decisiones. [26]

4.1.3. Estimación de armónicos

La cantidad de dispositivos electrónicos basados en semiconductores conectados a la red ha aumentado considerablemente durante los últimos años. Esto se debe a que la cantidad de cargas no lineales conectadas a la red ha aumentado. No solo las cargas, sino que cada vez los generadores utilizan más dispositivos basados en electrónica de potencia. La generación convencional basada en grandes generadores síncronos (onda senoidal, con baja distorsión armónica) están siendo sustituidos por generadores con interfaces basadas en electrónica de potencia (Converter-Interfaced Generators). Estos dispositivos introducen armónicos en el sistema eléctrico de potencia debido a las conmutaciones de los semiconductores. [26]

Algunos de los problemas más comunes en las redes eléctricas creados por la distorsión armónica son:

- Sobretensiones.
- Disparo de interruptores y apertura de circuitos inintencionadamente.
- Funcionamiento erróneo de equipos.
- Interferencias con los equipos de comunicación.
- Calentamiento de conductores.
- Problemas con los equipos de medición.
- Rotura en el aislamiento de equipos.

Para reducir los problemas creados por la distorsión armónica en la tensión de alimentación, es necesario introducir filtros especiales. Para diseñar estos filtros, es necesario determinar la magnitud y fase de los armónicos. Mediante el uso de redes neuronales, se consiguen métodos eficientes y rápidos de computación. Las RNAs son capaces de establecer y generalizar las relaciones entre los diferentes datos de entrada y detectar cuándo hay armónicos, su magnitud y su fase. También funcionan en tiempo real, que es una ventaja añadida en este caso.

Las redes neuronales más utilizadas son MLP con BackPropagation y la red neuronal de Fourier (Fourier Neural Network, FNN). Entre los dos métodos, el más preciso es el MLP (precisión del 95%). Comparándolo con otros métodos de cálculos de harmónicos como la Transformada Rápida de Fourier (Fast Fourier Transformation, FFT) o el algoritmo genético (GA), las redes neuronales son el método más fiable para la estimación de armónicos

4.1.4. Detección de faltas

La aparición de una falta es uno de los problemas más críticos para la seguridad y el buen funcionamiento de un sistema de distribución de energía eléctrica. Si ocurre una falta y no se actúa rápidamente para aislarla, un gran número de equipos pueden resultar gravemente dañados. [26]

La detección temprana de faltas y el aislamiento casi instantáneo de las mismas son fundamentales y el aislamiento casi instantáneo de la sección defectuosa son fundamentales para mantener la estabilidad del sistema. Para detectar faltas, se utiliza la información de la operación de los relés de protección y de los interruptores.

Las redes neuronales han demostrado ser el mecanismo más eficiente para la detección de faltas. Son sistemas con una mayor tolerancia a faltas que los métodos de detección de faltas convencionales. Esto ocurre debido a que tienen muchos más nodos de procesamiento, cada uno con sus conexiones locales. El malfuncionamiento de unos pocos nodos no afecta drásticamente al funcionamiento general del sistema. Además, pueden aprender y almacenar información sobre las faltas mediante memorización asociativa, y son capaces de diagnosticar asociativamente las faltas que ocurren en los sistemas de potencia. Predicen y localizan faltas con eficacias que rondan desde el 95% al 98%.

En cuanto a los tipos de RNA que se usan para detección de faltas, se usan varios tipos diferentes. La MLP para identificar el tipo y localización de la falta. La red de Kohonen

también se usa, pero es inferior a MLP en cuanto a precisión debido a que el tipo de aprendizaje que necesita es no supervisado. Respecto a los modelos RBF y BP, los BP tienen un mejor funcionamiento, pero los RBF tienen entrenamientos más rápidos. [27]

4.1.5. Evaluación de seguridad

Debido a las imposiciones de condiciones económicas, medioambientales y técnicas del sistema eléctrico de potencia, a veces es necesario hacer que la red funcione cerca de sus límites operacionales. Como resultado, la probabilidad de que ocurran perturbaciones o contingencias que lleven a una interrupción del servicio a los consumidores o al colapso del sistema es mayor. [26]

La seguridad de un sistema eléctrico es una característica de una red que garantiza un funcionamiento continuo en situaciones normales, e incluso si se producen algunas contingencias. Mediante la evaluación de la seguridad de un sistema eléctrico, se analiza que la red eléctrica, en cualquier punto de funcionamiento, sea capaz de soportar algunas contingencias. [29]

Con el objetivo de evitar cortes en el suministro, se pueden usar RNAs, ya que son una fuente de computación de confianza. Evaluar la seguridad de la red implica cierta cantidad de análisis de datos que se requiere para determinar si un sistema puede alcanzar un nivel particular de confianza y seguridad en los estados transitorio y dinámico para todas las probables contingencias o perturbaciones que puedan ocurrir. Para ello, es crucial que los métodos de computación sean rápidos, para poder solucionar en tiempo real los problemas de una forma segura, de confianza y económica.

Se pueden diferenciar dos tipos de evaluaciones de seguridad.

- Evaluación de seguridad estática (Static Security Assessment) Considera que, ante una perturbación, el sistema eléctrico se estabiliza en unas condiciones dentro de los límites de operación admisibles. Las restricciones aseguran que el sistema está adecuadamente en equilibrio, las tensiones de todos los buses dentro de sus límites aceptables, y que los límites térmicos de las líneas de transmisión no se sobrepasan. En el caso de que se violen las restricciones, podría resultar en un colapso general del sistema y apagones.

- Evaluación de seguridad dinámica (Dynamic Security Assessment). Es el análisis necesario para saber si un sistema eléctrico de potencia cumple ciertos criterios de seguridad y fiabilidad en los estados transitorio y estacionario para todas las posibles contingencias. También se lleva a cabo la gestión online, donde se toman medidas, proporciona información casi en tiempo real sobre el estado del sistema al operador o a los sistemas de control directamente.

También se utilizan redes neuronales para el reconocimiento de patrones, como la MLP, LVQ (Learning Vector Quantization), PNN (Probabilistic Neural Network) y ARTMAP (Adaptative Resonance Theory Mapping). [26]

El modelo MLP es el más utilizado por su buena adecuación a las características de los sistemas eléctricos de potencia. [27]

4.1.6. Estudio de estabilidad de voltaje

El análisis de estabilidad de tensión es una de las tareas más desalentadoras para las empresas de servicios de energía. En muchos casos, debido a la escasez de fondos, o por factores medioambientales, la expansión de los sistemas eléctricos de potencia ha sido limitada, haciendo que los sistemas eléctricos operen más cerca de sus límites de estabilidad. Las probabilidades de que haya problemas de estabilidad de la tensión u ocurra un colapso de tensión, por lo tanto, son mayores. [26]

Las RNA se han vuelto bastante populares entre los investigadores en los últimos años como una herramienta para la monitorización de la estabilidad del voltaje online.

El perceptron multicapa es el tipo de RNA más utilizado para los análisis de estabilidad de voltaje, debido a su éxito resolviendo problemas complejos y diversos. Se entrenan a sí mismas utilizando un algoritmo de BackPropagation.

Sin embargo, todavía existe cierto rechazo a utilizar RNA para estos usos. En general, la cantidad de datos de entrada de los sistemas eléctricos de potencia es inmensa. La relación funcional en sí cambia de una topología a otra, y limita el uso de las RNA. Para solucionar estos contratiempos, es necesario que sólo se utilicen los datos más importantes para obtener RNA más eficientes y compactas. Si los requerimientos de computación son altos, puede que sea necesario utilizar más de una RNA.

4.2. Implementación de RNAs

Las RNAs se llevan utilizando satisfactoriamente durante cuatro décadas. Al comienzo, los investigadores estaban obligados a construir su propio software para implementar redes neuronales, y finalmente, construir un hardware específico para ejecutar sus necesidades. [24]

La mayor parte de redes neuronales se implementan en software, debido a su menor coste de desarrollo. La implementación software ha evolucionado mucho a lo largo de los años, donde el resultado final ha sido la creación de muchas librerías para diferentes lenguajes de programación, que permiten crear redes neuronales fácilmente.

La implementación hardware de redes neuronales muestra muchas más posibilidades que la implementación software, sin embargo, hoy en día son menos utilizadas. La integración con los sistemas clásicos de computación es más complicada y cara que las implementaciones software, y también más difíciles de adaptar para la resolución de otros problemas. [30]

La mayoría de implementaciones software tienen como soporte hardware un ordenador personal o PC. Simplemente, es hardware que ejecuta el software en el que están programadas las redes neuronales. Los PC, sin embargo, son grandes, pesados y caros. El abandono de la implementación en un PC busca reducir el coste de la implementación, conseguir una mayor velocidad de procesamiento o simplemente usar implementaciones más sencillas.

En la década de los 80, se popularizó el uso de los circuitos integrados para aplicaciones específicas (Application Specific Integrated Circuits, ASIC), y se hizo un trabajo

considerable con el diseño e implementación de redes neuronales. Muchos de estos intentos por implementar RNAs en ASICs no fueron exitosos debido a que las RNAs implementadas en hardware no eran muy demandadas. [24]

Otro tipo de implementación hardware que se utilizó en esa década se basa en dispositivos lógicos programables (Programable Logic Devices, PLD). Permitían crear múltiples módulos lógicos hardware y enlazarlos formando sistemas más complejos. De esta forma, los PLD permiten desarrollar implementación hardware de neuronas en su base, y combinarlas formando una red. Esta estructura de PLD permite aprovecharse del procesamiento en paralelo que es necesario en las redes neuronales, y que no siempre es posible en otras implementaciones debido a el procesamiento pseudo-paralelo. [30]

En la década de los 90, apareció un nuevo tipo de dispositivo, el llamado matriz de puertas lógicas programable en campo (Field Programmable Gate Array, FPGA). La FPGA permite a los programadores reconfigurar las conexiones hardware entre las matrices de puertas lógicas, es decir, permite reconfigurar su uso. Es una alternativa más realista para la implementación de RNAs que la del uso de los ASIC o PLD.

Hoy en día hay muchas opciones de implementación software y hardware disponibles en el mercado.

4.2.1. Implementación software

En la Figura 29 se pueden observar los softwares más utilizados en la comunidad de desarrolladores de RNAs, según las encuestas realizadas en [24]. Dentro de "Otros softwares", entran programas como: Neuron y Python (3%), Mathematica, PyBrain, Nest and Brian (2%), Lens, PDP, Weka, NNPred, NeuroSolution, LVQPAK, SOM Toolbox, NNSYSID, Torch, GENESIS, Topographica, Oger, NetLab, AMORE, NEAT, MAPLE y libSVM (1%).

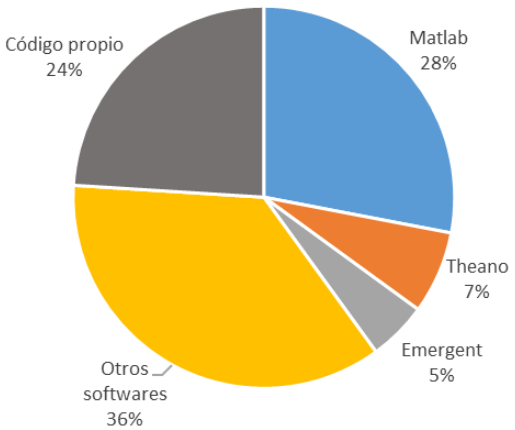

Figura 29 Software frecuente para el desarrollo de RNAs, 2013. [24]

Hay que mencionar que esta encuesta, realizada en 2013, muestra unos datos un tanto anticuados, ya que el sector de las redes neuronales, Deep Learning e inteligencia artificial

se está desarrollando rápidamente. Durante los últimos años, se han desarrollado programas que han tenido un éxito enorme entre las comunidades de desarrolladores de redes neuronales, como el software de Google de código abierto TensorFlow, o PyTorch desarrollado por Facebook.

A continuación, se muestran los programas más utilizados e interesantes para el desarrollo de RNAs.

4.2.1.1 MATLAB

MATLAB (MATrix LABoratory) es un sistema de cómputo numérico que ofrece un entorno de desarrollo integrado con un lenguaje de programación propio (lenguaje M). Está disponible para las plataformas Unix, Windows, macOS y GNU/Linux. [31]

Entre sus prestaciones básicas se hallan la manipulación de matrices, la representación de datos y funciones, la implementación de algoritmos, la creación de interfaces de usuario y la comunicación con programas en otros lenguajes y con otros dispositivos hardware. El paquete MATLAB dispone de dos herramientas adicionales que expanden sus prestaciones, a saber, Simulink (plataforma de simulación multidominio) y GUIDE (editor de interfaces de usuario). Además, se pueden ampliar las capacidades de MATLAB con las cajas de herramientas (toolboxes); y las de Simulink con los paquetes de bloques (blocksets).

MATLAB ofrece toolboxes especializadas con herramientas y funciones para administrar grandes conjuntos de datos para trabajar con machine learning, redes neuronales, deep learning, visión artificial y conducción autónoma. [32]

MATLAB permite desarrollar redes neuronales sin ser un experto. Se pueden poner en marcha rápidamente, crear y visualizar modelos de redes neuronales o desplegar modelos en servidores y dispositivos embebidos.

MATLAB permite integrar los resultados en sus aplicaciones existentes. Automatiza la implementación de sus modelos de redes neuronales en sistemas de empresa, clusters, nubes y dispositivos embebidos.

Deep Learning Toolbox proporciona un marco para diseñar e implementar redes neuronales profundas con algoritmos, modelos previamente entrenados y apps. [33] Permite entrenar:

- Redes neuronales convolucionales (ConvNet y CNN).
- Redes de memoria a corto plazo (LSTM) para realizar la clasificación y la regresión en imágenes, series temporales y datos de texto.
- Redes generativas antagónicas (GAN) y redes siamesas mediante diferenciación automática, bucles de entrenamiento personalizados y pesos compartidos.

Con la app Deep Network Designer, se puede diseñar, analizar y entrenar redes gráficamente. La app Experiment Manager ayuda a gestionar varios experimentos de deep learning, realizar un seguimiento de los parámetros de entrenamiento, analizar resultados y comparar código de diferentes experimentos. Puede visualizar las activaciones de capas y supervisar gráficamente el progreso del entrenamiento.

Puede intercambiar modelos con TensorFlow y PyTorch a través del formato ONNX e importar modelos de TensorFlow-Keras y Caffe. La toolbox soporta la transferencia de aprendizaje con DarkNet-53, ResNet-50, NASNet, SqueezeNet y muchos otros modelos previamente entrenados.

Se puede acelerar el entrenamiento en una estación de trabajo con una o varias GPU (con Parallel Computing Toolbox) o ampliar el alcance a clusters y nubes, incluidas las instancias de GPU de NVIDIA GPU Cloud y Amazon EC2 (con MATLAB Parallel Server).

4.2.1.2 TensorFlow

TensorFlow es una biblioteca de código abierto desarrollado por google, para el aprendizaje automático. Permite construir todo tipo de redes neuronales. Inicialmente, TensorFlow fue desarrollado para el uso interno en Google antes de ser publicado bajo licencia de código abierto en noviembre de 2015. Actualmente es utilizado tanto en investigación como en los productos de Google. [34] [35] [36]

Comenzó en 2011 bajo el nombre de DistBelief en 2011. Su uso creció rápidamente a través de compañías de Google. Gracias a varios científicos computacionales, se reconstruyó su código base en una biblioteca de grado aplicación más rápida y robusta, siendo TensorFlow el resultado. El nombre TensorFlow viene de las operaciones que las redes neuronales realizan sobre matrices multidimensionales de datos. Estas matrices son referidas como tensores.

Mientras la implementación de referencia se ejecuta en dispositivos aislados, TensorFlow puede correr en múltiple CPUs y GPUs (con extensiones opcionales de CUDA para informática de propósito general en unidades de procesamiento gráfico). TensorFlow está disponible para Windows, Linux, macOS, y plataformas móviles que incluyen Android e iOS. [37]

TensorFlow proporciona APIs estables de Python y C++, así como APIs no garantizadas compatibles con otros lenguajes.

Se centra en la simplicidad y la facilidad de uso, y puede utilizar una API intuitiva de alto nivel basada en Keras, y es desplegable en modelos en cualquier plataforma.

4.2.1.3 PyTorch

PyTorch es una biblioteca de aprendizaje automático de código abierto basada en Python. [38] [39] [40]

Permite dos características de alto nivel:

- Diseñar redes neuronales profundas.
- Realizar cálculos numéricos mediante la programación de tensores, con una fuerte aceleración mediante GPU. se pueden operar en una GPU de Nvidia compatible con CUDA.

Está basado en la biblioteca de Torch, que se utiliza para aplicaciones como visión artificial y procesamiento de lenguajes naturales.

La mayoría de los marcos como TensorFlow, Theano, Caffe y CNTK tienen una visión estática de las redes neuronales. Se construye una red neuronal, y se reutiliza la misma estructura una y otra vez. Para cambiar la forma en que la red se comporta, hay que empezar desde cero.

Con PyTorch, se utiliza una técnica llamada autodiferenciación en modo inverso, que permite cambiar la forma en que se comporta la red de forma arbitraria sin ningún tipo de retraso o sobrecarga.

4.2.1.4 Requisitos hardware para el entrenamiento de RNAs

Hoy en día, se pueden ejecutar redes neuronales casi con cualquier dispositivo. Hablando sobre requisitos hardware, la ejecución no es una fase tan crítica como el entrenamiento. Durante el entrenamiento, se van a tener que realizar una cantidad enorme de cálculos, por lo que el hardware donde se ejecute el entrenamiento es crucial para el tiempo que tarda para entrenarse la red. [41]

A la hora de elegir hardware para desarrollar redes neuronales, existen dos opciones, la utilización de servidores de grandes empresas pagando una cuota, o utilizar un PC.

Varias empresas (AWS, Azure, Google Cloud, etc.) permiten el alquiler de sus servidores, donde se crea un servidor a medida, configurando la cantidad de RAM, cuantas CPU se necesitan, y la tarjeta gráfica que se va a utilizar. No es una opción muy económica, y muchas veces no es la opción más eficiente.

Otra alternativa es utilizar un PC. Es probable que entrenar redes neuronales sencillas de un número bajo de neuronas sea viable en un ordenador portátil personal, pero si se quieren entrenar redes neuronales de mayor tamaño, es probable que el tiempo de entrenamiento sea exagerado, y que un ordenador portátil personal no sea capaz de realizar ese entrenamiento. Una opción es construir un PC con una configuración específica para el entrenamiento de redes neuronales. Esta opción supone un ahorro económico importante respecto a la utilización de servidores.

Los componentes críticos son principalmente la GPU, la CPU y la memoria RAM.

La tarjeta gráfica (Graphic Processing Unit, GPU), es el componente que va a realizar la mayoría de cálculos pesados cuando se entrena una red neuronal. Principalmente, estos cálculos incluyen convoluciones, multiplicaciones de matrices y funciones de activación. [42]

Hay 3 requisitos fundamentales a la hora de elegir una GPU:

- Velocidad de procesamiento, va correlacionado con la memoria de la GPU. La velocidad de procesamiento determina el número de operaciones que se realizan por segundo. Una GPU más rápida requiere menos tiempo de entrenamiento.

- Memoria, almacena toda la información utilizada durante el entrenamiento (pesos de conexiones, valores de los filtros de convolución, etc.). Gracias a tener esa información almacenada en la GPU, que es donde se realizan los cálculos, se puede afinar el entrenamiento para que funcione mejor.

- Se recomienda el uso de tarjetas gráficas de la marca NVIDIA, ya que la integración con softwares de desarrollo de redes neuronales es mejor.

La unidad central de procesamiento (Central Processing Unit, CPU) tiene un papel esencial en el entrenamiento de redes neuronales, ya que es la encargada de decodificar, normalizar y realizar otras técnicas de pre-procesamiento. Si la CPU no es lo suficientemente rápida, los datos tardarán en llegar a la GPU, desaprovechando la capacidad de procesado de datos de ésta, y ralentizando el entrenamiento.

La importancia de la CPU aumenta si el dispositivo no tiene la suficiente memoria RAM para almacenar todo el conjunto de datos. En ese caso, habría que almacenar parte de ellos en la memoria del disco duro, y en cada iteración la CPU se encargaría de leer los datos del disco duro. Esto haría que la carga de trabajo de la CPU aumente.

La memoria RAM se tiene que escoger teniendo en cuenta el tipo de modelos de redes neuronales que se quiere entrenar. Para entrenar modelos simples, con conjuntos de datos pequeños, permite cargar todos en la memoria RAM. Para entrenar modelos muy grandes, haría falta una cantidad de memoria muy grande. Una decisión razonable es caer en un punto medio, y mantener la RAM en el rango de 32-64 GB.

También hay varios requisitos menos importantes, como el disco duro, la placa base, la fuente de alimentación o la refrigeración.

El disco duro tiene que tener la velocidad de escritura y lectura de datos lo suficientemente alta para mantener la GPU ocupada todo el rato. También es importante que tenga la capacidad suficiente para almacenar todos los datos de entrenamiento. Los discos duros de estado sólido (SSD) son los más recomendados.

La refrigeración tiene un papel importante también. Cuando la CPU procesa datos, su temperatura sube. En el caso de que supere cierto umbral de temperatura (aproximadamente 90ºC), su velocidad de procesamiento se reduce para evitar que se siga calentando y prevenir daños. Si no se dispone de una refrigeración adecuada, el rendimiento de la CPU cae, y se puede convertir en un cuello de botella, alargando el tiempo de entrenamiento.

4.2.2. Plataformas hardware low cost para implementaciones software

Hay aplicaciones de RNAs donde es inviable introducir una implementación software que se ejecuta en un PC, por el tamaño, peso o coste. El avance tecnológico de los últimos años ha hecho posible la implementación software de RNAs en plataformas hardware que cuentan con un bajo precio, tamaño reducido, alta capacidad de procesamiento de datos y fácil diseño. Entre las múltiples opciones disponibles en el mercado, las siguientes plataformas son las más destacadas.

4.2.2.1 Arduino

Arduino es una plataforma de creación de electrónica de código abierto, la cual está basada en hardware y software libre, flexible y fácil de utilizar. Esta plataforma permite crear diferentes tipos de micro controladores de una sola placa a los que se les puede dar múltiples tipos de uso. [43]

El proyecto nació en 2003, cuando varios estudiantes del Instituto de Diseño Interactivo de Ivrea, Italia, con el fin de facilitar el acceso y uso de la electrónico y programación.

La placa cuenta con todos los elementos necesarios para conectar periféricos a las entradas y salidas del microcontrolador, y que puede ser programada tanto en Windows como macOS y GNU/Linux.

Arduino es un proyecto y no un modelo concreto de placa. En la Figura 30 se pueden observar varios modelos de placa que actualmente están a la venta. Compartiendo su diseño básico, se pueden encontrar con diferentes tipos de placas. Las hay de varias formas, tamaños y colores para a las necesidades del proyecto, y pueden ser más sencillas o con características mejoradas.

Figura 30 Diferentes placas de desarrollo Arduino. [43]

En la Tabla 4 se pueden ver los datos técnicos de la placa de desarrollo Arduino Uno.

Tabla 4 Características técnicas del Arduino One. [44]

Microcontrolador	ATmega328
Voltaje de operación	5V
Voltaje de entrada (recomendado)	7-12V
Voltaje de entrada (límites)	6-20V
Pines de E/S digitales	14 (de los cuales 6 proporcionan salida PWM)
Pines de entrada analógica	6
Corriente DC por pin de E/S	40 mA
Corriente DC para 3.3V Pin	50 mA
Memoria Flash	32 KB de los cuales 0,5 KB utilizados por el bootloader
SRAM	2 KB (ATmega328)
EEPROM	1 KB (ATmega328)
Velocidad de reloj	16 MHz

Los proyectos realizables con una placa Arduino son muy variables en cuanto a complejidad y temática se refiere. A continuación, se muestran varios proyectos realizados por la comunidad de desarrolladores. [45]

- Estación meteorológica.
- Acceso de seguridad.
- Alarma para el hogar.
- Control de persianas por voz.
- Automatización de jardines.
- Lector de huellas dactilares.
- Robótica.

4.2.2.2 Raspberry Pi

La Raspberry Pi es una serie de ordenadores de placa única u ordenadores de placa simple (SBC) de bajo coste desarrollado en el Reino Unido por la Raspberry Pi Foundation, con el objetivo de facilitar el acceso a la informática y la creación digital. En su esencia, son mini PC. Aunque el modelo original buscaba la promoción de la enseñanza de informática en las escuelas, este acabó siendo más popular de lo que se esperaba, hasta incluso vendiéndose fuera del mercado objetivo para usos como robótica. [46]

El hardware es un producto con propiedad registrada, pero se permite su uso libre tanto a nivel educativo como particular.

En cambio, el software sí es de código abierto, siendo su sistema operativo oficial una versión adaptada de Debian, denominada Raspberry Pi OS o Raspbian, aunque permite usar otros sistemas operativos, incluido una versión de Windows 10.

Figura 31 Raspberry Pi 4B.

Actualmente, hay varios modelos a la venta, siendo la Raspberry Pi Model 4B el modelo más reciente con mejores características (Tabla 5).

Tabla 5 Características técnicas de la Raspberry Pi 4B. [47]

SOC	Broadcom BCM2711
CPU	Procesador de cuatro núcleos a 1,5 GHz con brazo Cortex-A72
GPU	VideoCore VI
Memoria RAM	1/2/4GB LPDDR4
Conectividad	802.11ac Wi-Fi / Bluetooth 5.0, Gigabit Ethernet
Vídeo y Sonido	2 x puertos micro-HDMI que admiten pantallas de 4K@60Hz a través de HDMI 2.0, puerto de pantalla MIPI DSI, puerto de cámara MIPI CSI, salida estéreo de 4 polos y puerto de vídeo compuesto
Puertos	2 x USB 3.0, 2 x USB 2.0
Alimentación	5V/3A vía USB-C, 5V vía cabezal GPIO
Expansión	Cabezal GPIO de 40 pines

Los proyectos realizables con una Raspberry Pi son muchos y de características muy diferentes. A continuación, se muestran varios proyectos ya hechos por otros usuarios, fácilmente replicables y ampliables. [45] [48]

- Servidor web.
- Servidor VPN.
- Sistema de control para casas inteligentes.
- Sistema de monitorización de energía de una vivienda.
- Impresora 3D.
- Control de voz para abrir la puerta del garaje.
- Robótica.
- Sistemas de control de regado.
- Estación meteorológica.

4.2.2.3 Jetson Nano

El Jetson Nano es un ordenador de placa reducida diseñado por la marca NVIDIA, que esta principalmente enfocado a la implementación de múltiples redes neuronales en paralelo para aplicaciones de inteligencia artificial. [49]

Figura 32 Jetson Nano.

Sus características son las siguientes. [50]

- Tamaño reducido (69,6 x 45 mm).
- CPU ARM Cortex-A57 MPCore de 4.
- GPU Nvidia Maxwell con 128 núcleos CUDA.
- 4 Gb de RAM.
- 16 GB de almacenamiento.
- 4 puertos USB 3.0.

En cuanto al software, cabe destacar que Jetson Nano es compatible con los frameworks de IA más populares del mercado: TensorFlow, PyTorch, Caffe, Keras, MXNet, etc.

Presenta un consumo energético muy bajo, de 5W, lo que facilita su integración en robots y dispositivos para hogares inteligentes. En cuanto a sus usos, se puede utilizar para procesamiento de voz, traducción instantánea, reconocimiento de imágenes, manipulación de vídeos, etc.

4.2.2.4 Google Coral Dev Board

El Google Coral Dev Board es un ordenador de placa reducida desarrollado por la empresa Coral. Esta placa se ha desarrollado únicamente con el objetivo de realizar tareas de machine learning, que se puedan integrar en tareas de producción de manera rápida. Es un ordenador completo, que cuenta con las siguientes características. [51]

- SOM (system-on-module) equipado con una CPU ARM Cortex-A53 de cuatro núcleos.
- Un microcontrolador ARM Cortex-M4F.
- Una GPU Vivante GC7000 Lite, conectada al Edge TPU.
- 1 GB de RAM.
- 8 GB de memoria eMMC.
- Coprocesador criptográfico.
- Wi-Fi y Bluetooth 4.1.
- Conectores de audio, USB 2.0/3.0, DSI, MIPI-CSI, Ethernet, HDMI y 40 pines GPIO.

Figura 33 Google Coral.

El Módulo Edge TPU es una pequeña ASIC que se ha diseñado para proporcionar un gran rendimiento en aplicaciones destinadas al aprendizaje automático. Es un acelerador de inteligencia artificial, cuyo objetivo es llevar a cabo tareas de inferencia para permitir el funcionamiento de algoritmos de machine learning de forma local. El conjunto es capaz de realizar hasta 4 billones de operaciones por segundo (TOPS).

Es compatible con la distribución Debian de GNU/Linux, y permite desarrollar modelos de redes neuronales usando el paquete de software de aprendizaje automático TensorFlow Lite.

4.2.2.5 Aceleradores USB

Los aceleradores USB aceleran de forma asequible la velocidad de desarrollo de aplicaciones para inferencias basadas en redes neuronales profundas y simplifica la creación de prototipos a desarrolladores que trabajan en cámaras inteligentes, drones, robots para uso industrial y otros dispositivos. Su funcionamiento es muy sencillo, solo hay que conectarlo al puerto USB 3.0 del PC donde se vaya a utilizar. Los dispositivos se pueden acumular para trabajar en paralelo en proyectos que necesiten mayor potencia. Hay dos modelos que destacan en el mercado.

- Neural Compute Stick 2 es la segunda generación del dispositivo USB que Intel distribuyó para la creación de dispositivos de inteligencia artificial y aprendizaje profundo. Está basada en la unidad de procesamiento de visión (VPU) Intel Movidius Myriad X. Técnicamente, es capaz de realizar más de 4 billones de operaciones por segundo (TOPS).

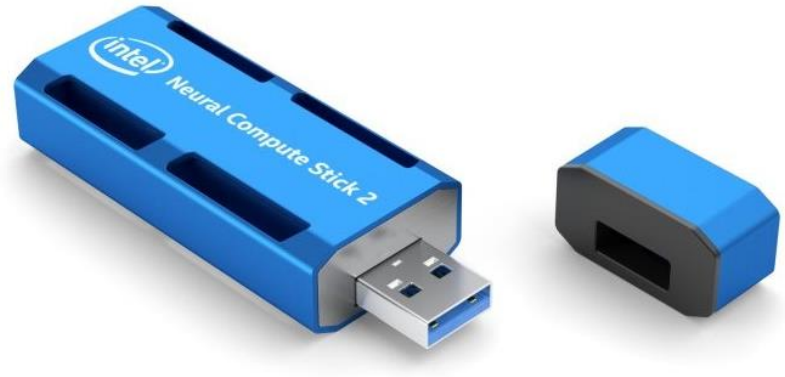

Figura 34 Intel Neural Compute Stick 2.

- Google Coral USB es un dispositivo USB que proporciona una TPU Edge como coprocesador para un ordenador. El coprocesador Edge TPU integrado es capaz de realizar cuatro billones de operaciones por segundo (TOPS).

Figura 35 Google Coral USB.

4.2.3. Implementación hardware

En la Figura 36 se puede observar las plataformas favoritas para la implementación hardware de la comunidad de desarrolladores de RNAs, según las encuestas realizadas en [24]. Las FPGA son la opción preferida, seguidas por los ASIC.

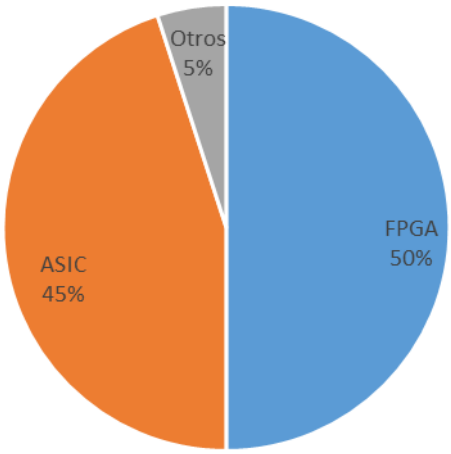

Figura 36 Plataformas preferidas para la implementación hardware de RNAs, 2013. [24]

A continuación, se muestran las opciones de implementación hardware más comunes.

4.2.3.1 ASIC

Un circuito integrado para aplicaciones específicas (Application-Specific Integrated Circuit, ASIC) es un circuito integrado hecho a la medida para un uso en particular, en vez de ser concebido para propósitos de uso general. Se usan para una función específica. [52]

Gracias a los avances en la reducción de tamaño de los chips y las herramientas de diseño, la complejidad y la funcionalidad de los ASIC ha crecido desde 5000 puertas lógicas a más de 100 millones. Los ASIC modernos incluyen desde procesadores de 32 bit, bloques de memoria RAM, ROM y otros tipos de módulos. Los ASIC se programan mediante lenguajes descriptores de hardware como Verilog o VHDL.

Las FPGA son la versión moderna de los ASIC, ya que permiten ser reprogramadas y ser utilizadas en muchas aplicaciones distintas.

4.2.3.2 CPLD

Los dispositivos lógicos programables complejos (Complex-Programmable Logic Devices, CPLD) extienden el concepto de un dispositivo lógico programable (Programmable Logic Device, PLD) a un mayor nivel. [53]

Los PLD son circuitos que tienen una función no establecida. Antes de poder utilizar el PLD, se tiene que programar.

Inicialmente se utilizaban ROMS para la integración de función fija, pero posteriormente se crearon las PLA (Programmable Logic Arrays), que son dispositivos compuestos por una serie de puertas lógicas AND que se encuentran entrelazadas a una serie de puertas OR, y cada una de ellas tiene su complementaria NOT para invertir el resultado. Permiten crear funciones de lógica combinacional. Otro tipo de lógica programable eran los llamados PAL (Programmable Array Logic), que tenían funcionamiento similar, pero eran menos versátiles que los PLA. Pero los PAL y los PLA pronto se quedaron desfasados y con el tiempo fueron apareciendo nuevos tipos de lógica programable. [54]

Un CPLD se forma con múltiples bloques lógicos, cada uno similar a un PLD. Los bloques lógicos se comunican entre sí utilizando una matriz programable de interconexiones. Un CPLD no es más que una serie de PLA como función de entrada, pero sus puertas OR no generan una salida, sino que su salida es distribuida a través de una matriz que conecta con otra serie de funciones PLA, las cuales pueden dar un resultado de salida o retroalimentar en dirección contraria.

En un CPLD, por tanto, todas las funciones lógicas posibles se encuentran codificadas en los diferentes PLA que se van combinando para conseguir así la función final que se quiere obtener. En general en diseño de nuevos procesadores los CPLD se utilizan tanto para simular la lógica combinacional como la función fija.

4.2.3.3 FPGA

Las matrices de puertas lógicas programables en campo (Field-Programmable Gate Arrays, FPGA) son una serie de dispositivos basados en semiconductores a base de matrices de bloques lógicos configurables o CLB, conectados mediante interconexiones programables. [55] [56]

Son el resultado de la convergencia de dos tecnologías diferentes, los PLD y los ASIC.

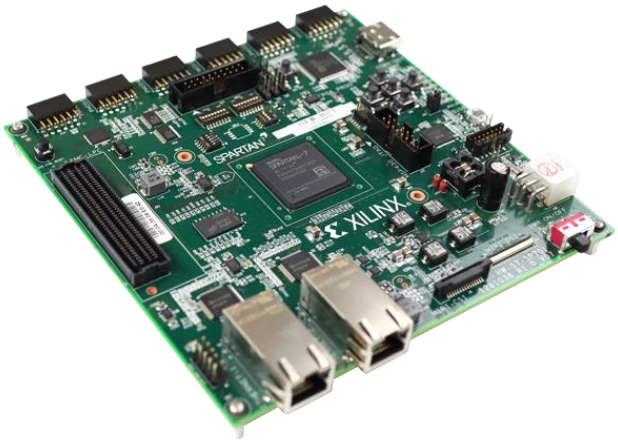

Figura 37 Spartan-7 SP701 FPGA.

Las FPGA están constituidas por miles o incluso millones de celdas que pueden emular cualquier puerta lógica o combinación de ellas, con salidas terminadas o no en flip-flops. Estas celdas están unidas mediante buses de interconexiones configurables como se

muestra en la Figura 38. También existen bloques lógicos de entrada/salida que realizan el interfaz con el exterior.

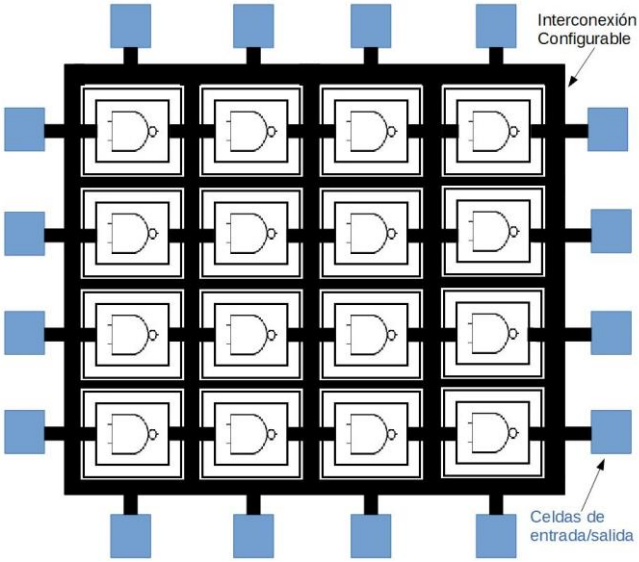

Figura 38 Buses de interconexiones configurables de una FPGA. [56]

La lógica programable de las FPGAs puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

Los principales fabricantes de FPGAs son Xilinx y Altera, pero existen también otros muchos fabricantes de FPGAs con diferentes características, como Lattice Semiconductor, Microsemi o QuickLogic.

Actualmente, existen FPGAs que integran memoria RAM, procesadores y motores DSP, por lo que están aumentando en potencia y complejidad, aumentando también el rango de aplicaciones en las que se pueden utilizar.

Estas se utilizan en aplicaciones similares a los ASIC, pero tienen una serie de ventajas e inconvenientes, que se muestran en la Tabla 6.

Tabla 6 Ventajas de las FPGAs sobre los ASIC.

Ventajas	Inconvenientes
• Son reprogramables. • Los costes de desarrollo y adquisición son mucho menores. • El tiempo de diseño y manufacturación es menor.	• Son más lentas. • Consumen mayor potencia. • No pueden realizar sistemas excesivamente complejos.

Hay que mencionar que los inconvenientes listados aquí se mantienen más por razones históricas que por motivos reales, ya que actualmente existen FPGAs lo suficientemente grandes para contener sistemas complejos, la velocidad de recursos de interconexión ha

aumentado exponencialmente y se ha reducido su consumo energético, sobre todo para FPGAs destinadas especialmente para dispositivos de bajo consumo.

Teniendo el mismo origen en los PLDs, es importante mencionar las diferencias entre las FPGA y los CPLD:

- Los FPGA están mejor preparados para simular hardware que depende del tiempo, ciclos de reloj, mientras que un CPLD está más pensado para lo que con circuitos combinacionales y que por tanto no dependen de una secuencia de pasos por ciclo.

- En un CPLD el tiempo que tarda una función sintetizada en el mismo será siempre el mismo, en un FPGA no. De ahí a que se utilicen los CPLD para diseñar circuitería combinacional.

- Los FPGA están pensados como productos de muy alta gama y por tanto con un alto coste. Los CPLD, al contrario.

- Los FPGA pueden funcionar a velocidades de reloj mucho más altas que un CPLD.

- Un FPGA necesita ser reprogramado cada vez que es encendido, ya que se desprograma al apagarse, un CPLD funciona como una ROM y no va a perder su configuración al apagarse el dispositivo. Esto significa que los FPGA se basan en memoria RAM en su arquitectura y los CPLD en memoria ROM.

A día de hoy los CPLD se suelen integrar dentro de placas de desarrollo FPGA, por lo que es común combinar ambos tipos de circuitería programable.

Las aplicaciones de las FPGAs en la actualidad son innumerables, ya que sus características técnicas pueden ser enfocadas a múltiples sectores como el sector aeroespacial, edición de audio, automoción, broadcast, electrónica, centros de tratamientos de datos, computación de alto rendimiento, sector industrial e industria médica.

4.2.4. Neurochips

Los neurochips son una implementación de redes neuronales en procesadores, y son capaces de implementar los algoritmos que éstas necesitan para funcionar. [2]

- Neurochips digitales. La mayoría de este tipo de neurochips están basados en semiconductores complementarios de óxido metálico (CMOS). Las ventajas de la tecnología digital son la calidad de fabricación, el almacenamiento de pesos en memoria RAM, y el diseño flexible. El procesador Micro Device MD1220 Neural Bit Slices es uno de los más básicos disponibles en el mercado.

  Una ventaja de estos chips es que se pueden asociar varios entre ellos para formar redes neuronales mayores, como se puede ver en el artículo [57]. Se basa en 4 procesadores MD1220 neural bit slice conectados a un PC. Se puede configurar el sistema como una única red o 4 independientes. Se implementan hasta 1024 neuronas, con 65.536 pesos internos de 16 bit de precisión.

- Neurochips analógicos. En este tipo de chips, los pesos se almacenan en forma de carga eléctrica en semiconductores. El chip de Intel ETANN (Electrically Trainable Analog Neural Network) es un chip analógico que incorpora una red MLP de 64 neuronas. La red no se puede entrenar en el chip, es preciso que sea entrenada previamente a la implementación. También son asociables entre ellos, habiendo sido utilizados para procesamiento de imágenes en tiempo real (12 chips ETANN) o en un sintetizador analógico de audio (8 chips ETANN).

- Neurochips híbridos. Este tipo de procesadores combinan las tecnologías analógicas y digitales para aprovechar las ventajas de ambas. Se aprovecha la velocidad de procesamiento interna de los chips analógicos, y los pesos se establecen y guardan de forma digital.

- Neurochips basados en RAM. Este tipo de tecnología se basa en elementos de procesamiento o neuronas que solo tienen entradas y salidas binarias, sin pesos entre los nodos. Las funciones neuronales están almacenadas en tablas de seguimiento. Este tipo de redes neuronales se puede entrenar muy rápidamente. En vez de seguir un entrenamiento tradicional ajustando los pesos, se entrenan cambiando los contenidos de las tablas de seguimiento.

4.2.5. Placas de código abierto híbridas con FPGA

Existen proyectos que intentan aprovechar las ventajas de las FPGA y de las placas de desarrollo de código abierto. Entre ellas, destacan el Arduino MKR Vidor 4000 y las placas Papilio.

4.2.5.1 Arduino MKR Vidor 4000

El Arduino MKR Vidor 4000 (Figura 39), consiste de una placa Arduino, que incluye una FPGA Intel Cyclone 10CL016. La programación se realiza mediante el software Arduino IDE, y se programa en el lenguaje de bajo nivel VHDL. Mediante esta combinación de Arduino y FPGA, se le añade a la complejidad de las FPGA la facilidad de uso y programación de Arduino. [58]

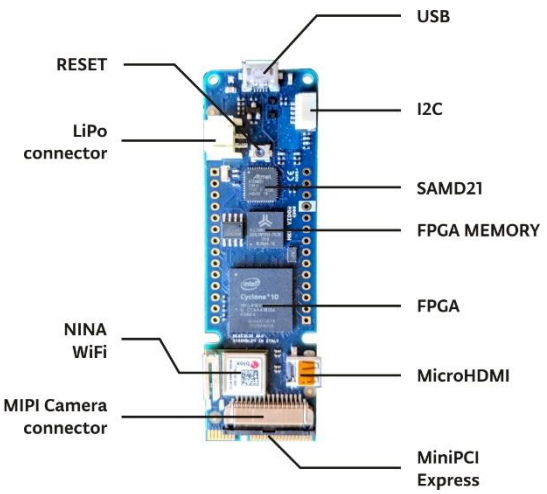

Figura 39 Arduino MKR Vidor 4000. [58]

Entre algunas de sus características, contiene 16.000 elementos lógicos, 504 Kb de RAM, 56 multiplicadores 18×18 de hardware para aplicaciones DSP de alta velocidad, etc. Los pines de la FPGA pueden dar salidas de hasta 150 MHz, y también se pueden configurar como puertos de comunicaciones comunes como UART, I2C y SPI.

Se pueden realizar operaciones DSP de alta velocidad para el procesamiento de audio y video. Por lo tanto, el Vidor incluye un conector Micro HDMI para salida de audio y video, y un conector de cámara MIPI para entrada de video.

Se pueden crear una serie de salidas PWM de alta frecuencia dedicadas, un mezclador de sonido digital, una máquina de superposición de video…. Los proyectos realizados por otros desarrolladores, como con todas las placas de Arduino, son muchos. La principal limitación con esta placa es la cantidad de puertas lógicas necesarias.

4.2.5.2 Papilio

Papilio es un proyecto inspirado y basado en Arduino, es una placa de desarrollo Open Source con una FPGA. La FPGA que integra permite tanto ejecutar código en C++ de Arduino, como programar periféricos VHDL. Se programa mediante el software Papilio Designlab IDE, que está basado en el IDE Arduino. Mediante la incorporación de librerías, se facilita el diseño de proyectos basados en FPGA. [59]

Hay varios tipos de placas: Papilio One (Figura 40), Papilio Duo, Papilio Pro, y varias ampliaciones. Si se quiere mejorar las características de la placa, existen otros elementos llamados “alas”, que se encajan con la placa y amplían las características.

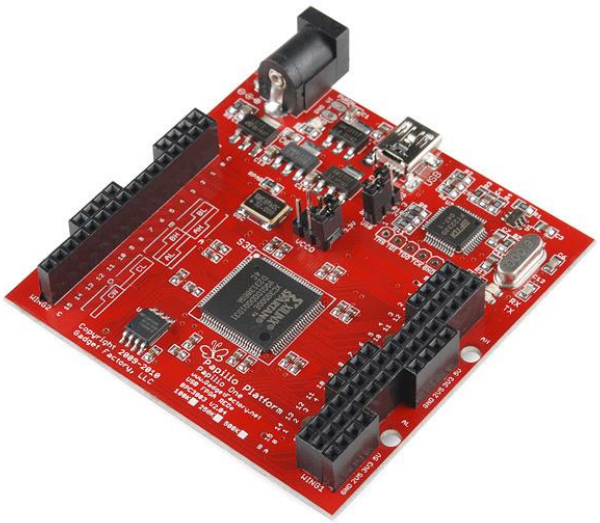

Figura 40 Papilio One.

A continuación, se listan las características de la placa Papilio One:

- FPGA Xilinx Spartan 3E.
- 4Mbit SPI.
- Conexión USB de dos canales para comunicaciones serie y JTAG.
- Cuatro rieles de alimentación independientes a 5 V, 3,3 V, 2,5 V y 1,2 V.
- Alimentación suministrada por un conector de alimentación o USB.
- Voltaje de entrada (recomendado): 6.5-15V.
- 48 entradas/salidas.

5. ANÁLISIS DE ALTERNATIVAS

En este apartado se realiza una comparativa de las diferentes plataformas de implementación de redes neuronales mencionadas.

5.1. FPGA vs PC

A diferencia de una CPU de un ordenador, es difícil definir la potencia de cálculo de un FPGA, dado que es algo totalmente distinto a un procesador como el que podemos encontrar en un Arduino, en un PC o incluso en un mini ordenador como Raspberry PI, Google Coral y Jetson Nano . [60]

Las FPGA destacan en la realización de tareas en paralelo, y por un control extremadamente fino del tiempo y el sincronismo de las tareas.

Una vez programado, el FPGA constituye físicamente un circuito. En general, como se ha comentado, un FPGA es más lento que el ASIC equivalente. Por otro lado, aunque los FPGA normalmente incorporan un reloj para la elaboración de tareas síncronas, en algunas de las tareas la velocidad es independiente del reloj, y están determinado por la velocidad de los componentes electrónicos que lo forman. En cualquier caso, las FPGA son una herramienta muy potente y lo suficiente diferentes del resto para ser interesantes por sí mismos.

Las FPGAs no sólo son muy difíciles de programar, incluso con las herramientas actuales que facilitan el proceso. También es necesario conocer a fondo el hardware que se utiliza. Aunque una FPGA cuesta sólo unos pocos dólares, algunas placas de diseño son relativamente caras, y, una vez realizado el diseño, no es tan fácil de cambiar sus características como en el resto de plataformas. [61]

Respecto a las diferencias del procesamiento de datos mediante una FPGA y una CPU, la CPU de un ordenador procesa una larga secuencia de pequeños pasos aritméticos. Cuanto más rápido se ejecute esta secuencia, más rápido se ejecutará el programa.

Una FPGA, en cambio, trabaja en paralelo. Una operación en la entrada puede provocar una reacción en cadena repartida entre muchas puertas, ejecutando así muchos cálculos complejos en un solo paso.

La CPU es flexible. Si se cambia el programa, la funcionalidad cambia en consecuencia. La funcionalidad de una FPGA es fija debido al cableado pre-programado. Durante el arranque, la tabla de conexiones se carga en la FPGA, después de lo cual permanece inalterada.

Otra diferencia importante es el tamaño de los números binarios. Una CPU tiene un tamaño fijo. Toda la arquitectura utiliza números de 16, 32 o 64 bits. Los números en una FPGA son flexibles. Como todo el cableado del chip funciona a nivel de bits, no hay problema si, por ejemplo, se necesitan 21 bits.

Una aplicación de implementar redes neuronales en FPGA que es muy superior a las plataformas basadas en CPU son cuando se tienen que integrar en sistemas con baterías, ya que la FPGA tiene una eficiencia energética superior en todos los sentidos, como se puede observar en la Figura 41.

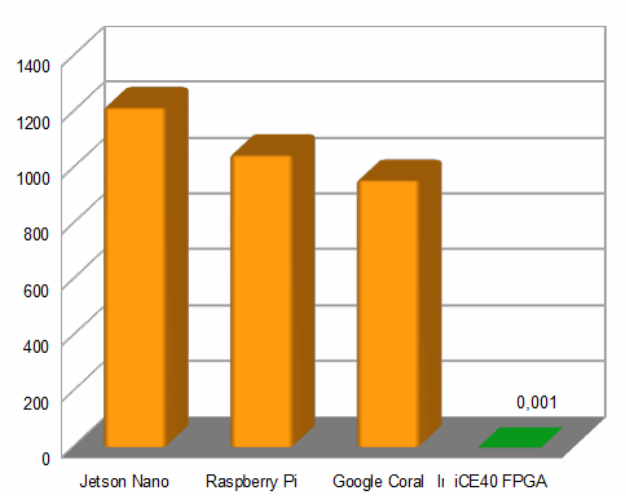

Figura 41 Consumo energético en mA de diferentes plataformas. [61]

En la Tabla 7 se realiza una comparación de las características de una implementación hardware (enfocada en una FPGA) y una implementación software en una plataforma hardware (enfocada en un mini PC), mediante una ponderación en función de cada característica.

Tabla 7 Comparativa implementación hardware vs software.

	FPGA	Mini PC
Velocidad de procesado	++++	+++
Tamaño de las redes implementables	+++	++++
Flexibilidad	+++	++++
Facilidad de la puesta en marcha	+	+++
Facilidad de programación de redes	+	+++
Coste económico	+++	+++
Consumo energético	+	+++
Tamaño	+	++
Acceso al material	NO	SI

Siendo los valores dados: ++++muy alto, +++ alto, ++ medio, + bajo.

5.2. Arduino vs Mini PC

Comparar un Arduino con una Raspberry Pi, el Google Coral o un Jetson Nano es complicado, puesto que, en términos de hardware, son cosas diferentes, como se puede apreciar en la Tabla 8. Raspberry Pi es un mini PC, mientras que Arduino es una micro

controladora. La principal diferencia entre ambas plataformas, es que Arduino solo es capaz de ejecutar un programa a la vez en bucle, mientras que con los minis PC se puede disponer del procesamiento en paralelo, es decir, se puede hacer lo mismo que con un ordenador normal. [62]

Tabla 8 Comparación de características entre Arduino y Raspberry Pi.

Característica	Raspberry Pi 4	Arduino Uno
SoC	BCM2711	ATmega328
CPU	Quad Cortex-A72 @ 1.5 GHz	16 MHz
Set de instrucciones	ARMv8	Arduino IDE
GPU	VideoCore VI	ATmega328
RAM	1GB, 2 GB, 4 GB (dependiendo del modelo)	2 KB
Almacenamiento	MicroSD	EEPROM 1 KB
Ethernet	10/100/1000	No
Wireless	802.11ac, Bluetooth 5.0, BLE	No
Salidas de vídeo	2 × micro-HDMI	No
Salidas de audio	HDMI / Auriculares	No

En cuanto a las características técnicas necesarias para integrar redes neuronales, como potencia de procesamiento, almacenamiento, y memoria RAM, Arduino no puede competir con los mini PC. Simplemente, está enfocado a otro uso, como podría ser la gestión de sensores o programar acciones de un robot autómata.

Hay varias razones que hacen que Arduino sea una plataforma interesante. La primera, es su muy reducido precio. A diferencia del resto de plataformas, es un dispositivo que se puede desconectar de la corriente sin ningún miramiento, y cuando se vuelva a conectar a la alimentación, se empezará a ejecutar el código desde el principio. En el resto de plataformas hay que seguir un procedimiento especial para evitar que se corrompan los datos ante un apagón inesperado. También tiene una gran cantidad de pines de entrada y salida de datos analógicos y digitales.

En la Figura 42 se comparan varias redes neuronales de procesamiento de imágenes y visión artificial implementadas en varias combinaciones de los mini PC. El ensayo muestra la velocidad de procesado de imágenes que tiene cada mini PC en fotogramas por segundo (FPS). Se puede observar cómo en función de las diferentes redes, las cuatro opciones que destacan son la Raspberry Pi 4B más el Intel Neural Stick 2, la Raspberry Pi 4B más el Google Coral USB, El Jetson nano y el Google Coral. En función de las características de la red, el rendimiento de una es mejor que otra. [63]

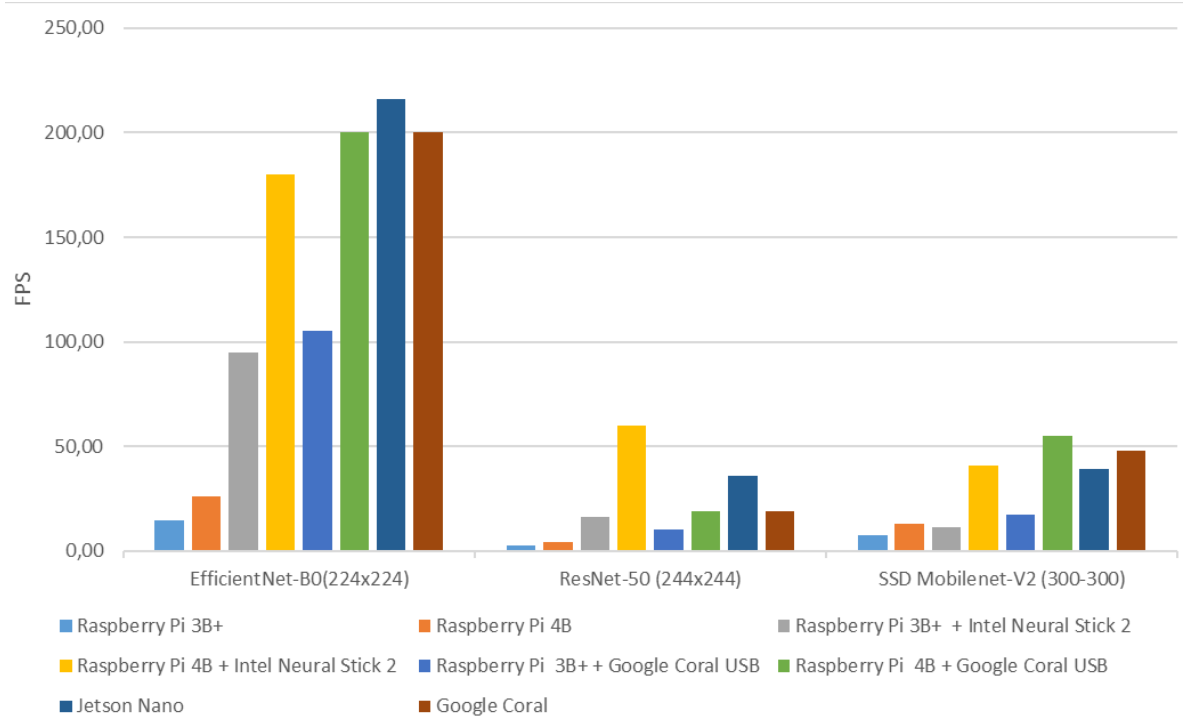

Figura 42 Comparación de mini PC con diferentes redes neuronales para el procesamiento de imágenes. [62]

En conclusión, se puede destacar lo siguiente. [64]

- La Raspberry Pi 4B por su precio, gran cantidad de tutoriales y ayuda disponible en internet. La desventaja que presenta con respecto al resto al no tener GPU o TPU se puede compensar añadiéndole un acelerador USB.

- El Jetson Nano por su flexibilidad. Tiene un buen rendimiento, y puede ejecutar cualquier cosa que ejecutan los PC normales. Además, destaca también por tener un software bien diseñado y robusto.

- Google Coral Dev Board por el buen rendimiento energético, debido a que monta un chip muy reciente. Dependiendo del tipo de red, puede ser más rápido incluso que el Jetson Nano.

En la Tabla 9 se resume la comparativa realizada entre las diferentes plataformas hardware low cost para la implementación software de redes neuronales.

Tabla 9 Comparativa de implementación en diferentes plataformas hardware low cost.

	Arduino	Raspberry Pi	Jetson Nano	Google Coral
Velocidad de procesado	+	++	+++	++++
Tamaño de las redes implementables	+	+++	++++	++++
Flexibilidad	+	+++	+++	+++
Facilidad de la puesta en marcha	+++	+	+	+
Facilidad de programación de redes	++	+++	+++	+++
Coste económico	+	+	++	++
Consumo energético	++	+++	++	++
Tamaño	+	+	++	++
Acceso al material	SI	SI	NO	NO

Siendo los valores dados: ++++muy alto, +++ alto, ++ medio, + bajo.

6. DESCRIPCIÓN DE LA SOLUCIÓN ADOPTADA

6.1. Implementación en Arduino

En este apartado se va a realizar una implementación software en la plataforma Arduino. Se ha optado utilizar esta plataforma, debido a su bajo coste, su facilidad de programación, y buen funcionamiento general. Los objetivos de esta implementación son los siguientes.

- Analizar la velocidad de respuesta de la red. Como se ha mencionado en apartados anteriores, Arduino no destaca con respecto a otras implementaciones por su velocidad de procesamiento de datos, por lo que un parámetro interesante a analizar es la velocidad de respuesta de la red.

- Analizar cuál es el máximo tamaño de red implementable en Arduino.

6.1.1. Problema a resolver

El objetivo de la red neuronal va a ser convertir los 7 segmentos de un display led numérico a binario. Las diferentes combinaciones posibles del display se muestran en la Figura 43. A cada combinación, le corresponde su número en binario, como se muestra en la Tabla 10. De este modo, se podría conseguir una especie de sistema de reconocimiento de dígitos rudimentario.

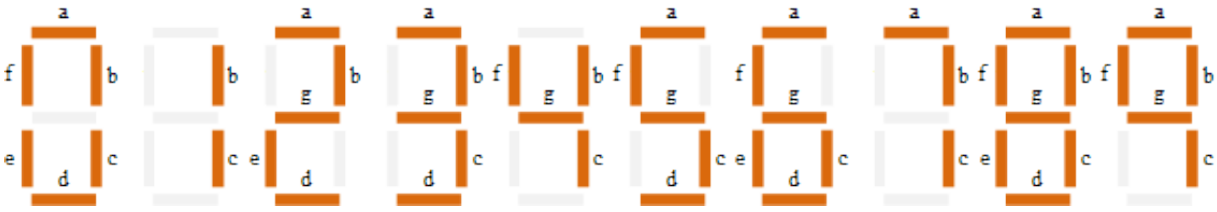

Figura 43 Diferentes combinaciones de dígitos.

En la Figura 44 se puede ver un ejemplo de lo que la red neuronal va a convertir.

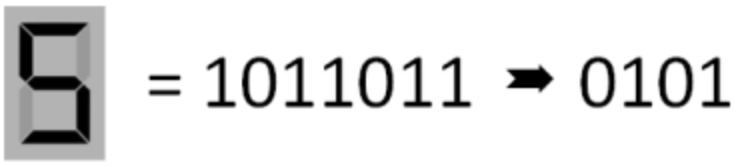

Figura 44 Ejemplo de conversión.

Tabla 10 Tabla de verdad.

Dec	a	b	c	d	e	f	g	Out 1	Out 2	Out 3	Out 4
0	1	1	1	1	1	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	1
2	1	1	0	1	1	0	1	0	0	1	0
3	1	1	1	1	0	0	1	0	0	1	1
4	0	1	1	0	0	1	1	0	1	0	0
5	1	0	1	1	0	1	1	0	1	0	1
6	0	0	1	1	1	1	1	0	1	1	0
7	1	1	1	0	0	0	0	0	1	1	1
8	1	1	1	1	1	1	1	1	0	0	0
9	1	1	1	0	0	1	1	1	0	0	1

6.1.2. Tipo de red neuronal utilizada

El tipo de red neuronal utilizada es una MLP con BackPropagation. Como se puede ver en la Figura 45, la red tiene 7 neuronas en la capa de entrada, una capa oculta de 8 neuronas, y una capa de salida de 4 neuronas. Una red de estas dimensiones es de las más grandes de se pueden diseñar en Arduino.

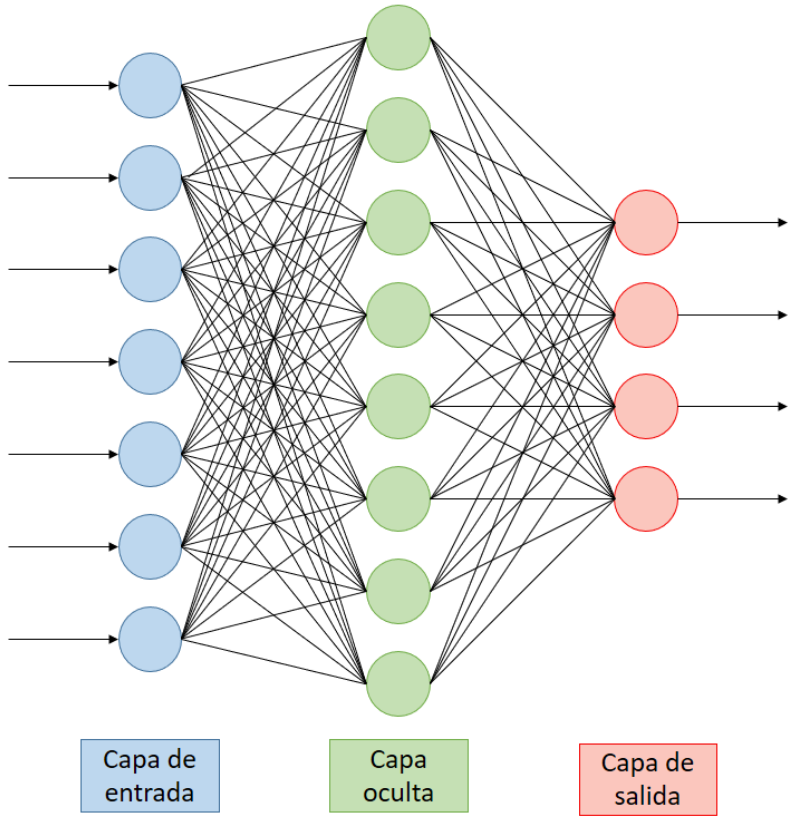

Figura 45 Red MLP propuesta.

6.1.3. Montaje del Arduino

Para realizar la programación y el montaje del Arduino que se muestra en la Figura 46, se ha utilizado el siguiente material.

- Un ordenador con el software de Arduino IDE instalado (fuente página oficial Arduino). Éste se va a utilizar tanto como para programar el Arduino, como para visualizar el entrenamiento y el funcionamiento de la red neuronal, e introducir los datos necesarios por el Serial Port. [65]
- Un Arduino Uno. Será el encargado de ejecutar la red neuronal programada.
- Cable de conexión de puerto serie.

Figura 46 Montaje del Arduino.

Para desarrollar esta red neuronal, se ha utilizado como base el trabajo realizado en [66].

6.1.4. Programación del Arduino

A continuación, se explica el funcionamiento de la red neuronal. La totalidad del código utilizado se incluye en el ANEXO Nº1: CÓDIGO Y PROGRAMACIÓN.

Se definen los siguientes parámetros.

- PatternCount: número de elementos de entrenamiento o líneas de la Tabla 10.
- InputNodes: número de neuronas de la capa de entrada.
- OutputNodes: número de neuronas de la capa de salida.
- HiddenNodes: número de neuronas de la capa oculta.
- LearningRate: tasa de aprendizaje.
- Momentum: momento, ajusta cuánto afectan a la iteración actual los resultados de la iteración anterior.
- InitialWeightMax: establece un valor máximo para los pesos iniciales.
- Success: umbral de éxito en el que se dice que la red ha resuelto el conjunto de entrenamiento.

Como concepto general, HiddenNodes, LearningRate, Momentum y InitialWeightMax trabajan juntos para optimizar la eficacia y velocidad del aprendizaje de la red, mientras que se minimizan ciertos problemas que se encuentran en el diseño de redes neuronales.

Un valor más bajo de la tasa de aprendizaje (LearningRate) da lugar a un proceso de entrenamiento más lento, pero reduce la probabilidad de que la red entre en una oscilación en la que sobrepasa continuamente la solución del problema de entrenamiento y nunca alcanza el umbral de éxito. En este caso, está fijado en 0,3. Para redes grandes y muy complejas (mucho más grandes de lo que podríamos construir en el Arduino Uno), el valor es a menudo muy bajo, del orden de 0.01.

El momento (Momentum) suaviza el proceso de entrenamiento, añadiendo una parte de la retropropagación anterior en la retropropagación actual. El impulso sirve para ayudar a prevenir un fenómeno en el que la red converge en una solución que es buena pero no la mejor, también conocido como convergencia en el mínimo local. Los valores del impulso deben estar entre 0 y 1.

El número de neuronas de la capa oculta afecta a la velocidad con la que se puede entrenar una red, a la complejidad de los problemas que puede resolver la red y puede ayudar a evitar la convergencia en el mínimo local. Es conveniente tener al menos tantas neuronas ocultas como neuronas de salida, y es posible que desee añadir más. El inconveniente de un gran número de neuronas ocultas es el gran número de pesos que hay que almacenar.

Los pesos iniciales, aunque sean aleatorios, deben ser relativamente pequeños. El valor de InitialWeightMax en la configuración escogida es 0.5. Esto establecerá todos los pesos iniciales entre -0.5 y 0.5, que son valores adecuados para comenzar el entrenamiento.

Los valores ideales para estos parámetros varían mucho dependiendo de los datos de entrenamiento y realmente no hay una forma determinada para conseguir el funcionamiento óptimo de la red neuronal. Estos valores se han establecido a base de prueba y error.

El valor final en la sección de configuración, Éxito, establece el nivel de error en el sistema cuando el conjunto de entrenamiento se considerará aprendido. Es un número muy pequeño mayor que cero. La naturaleza de este tipo de red es que el error total del sistema se aproximará a cero, pero nunca lo alcanzará.

Como se ha mencionado previamente, la red escogida, que cuenta con 7 entradas, 8 neuronas ocultas y 4 salidas, es la más grande que se puede implementar en la SRAM de 2K del Arduino Uno. Al compilar el programa, ya avisa de que se deja poca memoria libre para las variables locales, y que esto puede producir problemas de estabilidad. Desafortunadamente, no hay ninguna advertencia si te quedas sin memoria en el Arduino. Simplemente, el funcionamiento del código es erróneo.

La estrategia básica para la implementación de una red neuronal como un programa en C++ es:

- Establecer un marco de matrices de datos para guardar los pesos, y demás datos necesarios, a medida que las señales se propagan hacia adelante.

- Al acabar, los errores se alimentan hacia atrás a través de la red. Una secuencia de bucles FOR encadenados recorren las matrices realizando los cálculos necesarios a medida que se ejecuta el algoritmo de retropropagación de los errores.

A continuación, se expone el funcionamiento del programa a grandes rasgos. Consta de dos fases principales:

- Fase de entrenamiento. Se ejecuta una vez al encender el Arduino.
    - Elección del tipo de función de transferencia a utilizar, lineal o sigmoide.

- Inicialización de las matrices. Los pesos se establecen en números aleatorios. También hay dos matrices adicionales que contienen valores de cambio que se usan en la retropropagación. Inicialmente se establecen como ceros.
- Comienza un bucle grande, que ejecuta el sistema a través del conjunto completo de datos de entrenamiento.
- En cada iteración, el orden en el que se ejecutan los conjuntos de entrenamiento se hace aleatorio para reducir la oscilación y evitar la convergencia en los mínimos locales.
- Se calculan las activaciones de la capa oculta, las activaciones de la capa de salida, y el error (Ec. 7).
- Se retropropaga el error a la capa oculta.
- Se actualizan los pesos.
- Si el error del sistema es mayor que el umbral de éxito, se ejecuta otra iteración de los datos de entrenamiento.
- Si el error del sistema es menor que el umbral de éxito, se interrumpe el bucle, finaliza el entrenamiento, y envía los datos al terminal serie.
- Cada 1000 ciclos, envía los resultados de una iteración del conjunto de entrenamiento al terminal serie.

- Fase de ejecución. Se ejecuta en bucle sin fin al acabar la fase de entrenamiento.
  - A través del terminal serie, se piden los nuevos datos de entrada.
  - Con el modelo de la red ya entrenado, se calculan las activaciones de la capa oculta y las activaciones de la capa de salida, y se muestra el resultado por el serial.

6.1.5. Prueba de la red neuronal y validación

En la Figura 47 se muestran unas capturas de pantalla del serial por el que se controla la red neuronal. Tras escoger el tipo de función de transferencia que va a utilizar la red, comienza el entrenamiento. Se observa cómo, antes de ser entrenada, la red no da la salida deseada, comete un error muy grande.

```
<Arduino is ready>
11:38:39.307 -> Elige el tipo de función de transferencia
11:38:39.376 -> 1.- Lineal
11:38:39.376 -> 2.- Sigmoidea
11:38:43.480 -> Función de transferencia elegida  2.- Sigmoidea
11:38:43.515 ->
11:38:43.515 ->
11:38:43.515 -> Initial/Untrained Outputs:
11:38:43.549 ->
11:38:43.549 ->   Training Pattern: 0
11:38:43.583 ->   Input 1 1 1 1 1 1 0   Target 0 0 0 0   Output 0.44467 0.54465 0.64629 0.59062
11:38:43.651 ->   Training Pattern: 1
11:38:43.686 ->   Input 0 1 1 0 0 0 0   Target 0 0 0 1   Output 0.43730 0.57082 0.62818 0.54071
11:38:43.790 ->   Training Pattern: 2
11:38:43.790 ->   Input 1 1 0 1 1 0 1   Target 0 0 1 0   Output 0.47191 0.53619 0.62159 0.57705
11:38:43.891 ->   Training Pattern: 3
11:38:43.926 ->   Input 1 1 1 1 0 0 1   Target 0 0 1 1   Output 0.44053 0.56457 0.61946 0.56846
11:38:43.993 ->   Training Pattern: 4
11:38:44.027 ->   Input 0 1 1 0 0 1 1   Target 0 1 0 0   Output 0.48759 0.55217 0.59914 0.57633
11:38:44.130 ->   Training Pattern: 5
11:38:44.130 ->   Input 1 0 1 1 0 1 1   Target 0 1 0 1   Output 0.43498 0.58118 0.62702 0.55514
11:38:44.233 ->   Training Pattern: 6
11:38:44.267 ->   Input 0 0 1 1 1 1 1   Target 0 1 1 0   Output 0.47105 0.56996 0.62352 0.57500
11:38:44.336 ->   Training Pattern: 7
11:38:44.370 ->   Input 1 1 1 0 0 0 0   Target 0 1 1 1   Output 0.41677 0.56409 0.63963 0.52806
11:38:44.438 ->   Training Pattern: 8
11:38:44.473 ->   Input 1 1 1 1 1 1 1   Target 1 0 0 0   Output 0.47190 0.53902 0.62334 0.59708
11:38:44.542 ->   Training Pattern: 9
11:38:44.542 ->   Input 1 1 1 0 0 1 1   Target 1 0 0 1   Output 0.46572 0.54859 0.61231 0.56225
11:38:44.679 ->
11:38:44.679 -> TrainingCycle: 1  Error = 5.50976
11:38:44.713 ->
11:38:44.713 ->   Training Pattern: 0
11:38:44.748 ->   Input 1 1 1 1 1 1 0   Target 0 0 0 0   Output 0.19260 0.31891 0.28117 0.59629
11:38:44.816 ->   Training Pattern: 1
11:38:44.851 ->   Input 0 1 1 0 0 0 0   Target 0 0 0 1   Output 0.18523 0.35165 0.27874 0.56925
11:38:44.919 ->   Training Pattern: 2
11:38:44.954 ->   Input 1 1 0 1 1 0 1   Target 0 0 1 0   Output 0.19902 0.30246 0.25101 0.58516
11:38:45.057 ->   Training Pattern: 3
11:38:45.057 ->   Input 1 1 1 1 0 0 1   Target 0 0 1 1   Output 0.17577 0.32822 0.24638 0.59152
11:38:45.161 ->   Training Pattern: 4
11:38:45.196 ->   Input 0 1 1 0 0 1 1   Target 0 1 0 0   Output 0.22035 0.33171 0.24162 0.59238
11:38:45.264 ->   Training Pattern: 5
11:38:45.299 ->   Input 1 0 1 1 0 1 1   Target 0 1 0 1   Output 0.17422 0.34809 0.24962 0.57391
11:38:45.367 ->   Training Pattern: 6
11:38:45.401 ->   Input 0 0 1 1 1 1 1   Target 0 1 1 0   Output 0.20790 0.34946 0.26171 0.57969
11:38:45.504 ->   Training Pattern: 7
11:38:45.504 ->   Input 1 1 1 0 0 0 0   Target 0 1 1 1   Output 0.16901 0.33458 0.27781 0.55921
11:38:45.607 ->   Training Pattern: 8
11:38:45.641 ->   Input 1 1 1 1 1 1 1   Target 1 0 0 0   Output 0.20784 0.31023 0.25360 0.60034
11:38:45.710 ->   Training Pattern: 9
11:38:45.710 ->   Input 1 1 1 0 0 1 1   Target 1 0 0 1   Output 0.20120 0.31847 0.24275 0.58126
```

Figura 47 Captura de la ejecución del código en Arduino.

En la Tabla 11 se muestra el resumen de los diferentes ciclos de entrenamiento que realiza la red hasta alcanzar el error mínimo deseado (se ha definido en 0.0004).

Tabla 11 Resumen de los ciclos de entrenamiento de la red.

Ciclo de entrenamiento	Error	Tiempo
1	5.50976	00.00 s
1000	0.00268	99.48 s
2000	0.00124	99.54 s
3000	0.00079	99.68 s
4000	0.00058	99.85 s
5000	0.00046	100.01 s
5653	0.00040	65.767 s
Tiempo total de entrenamiento		09:24.33 s

Se puede observar que el Arduino utiliza aproximadamente 100 segundos para realizar 1000 iteraciones. El tiempo total de entrenamiento es de 09 minutos 24 segundos.

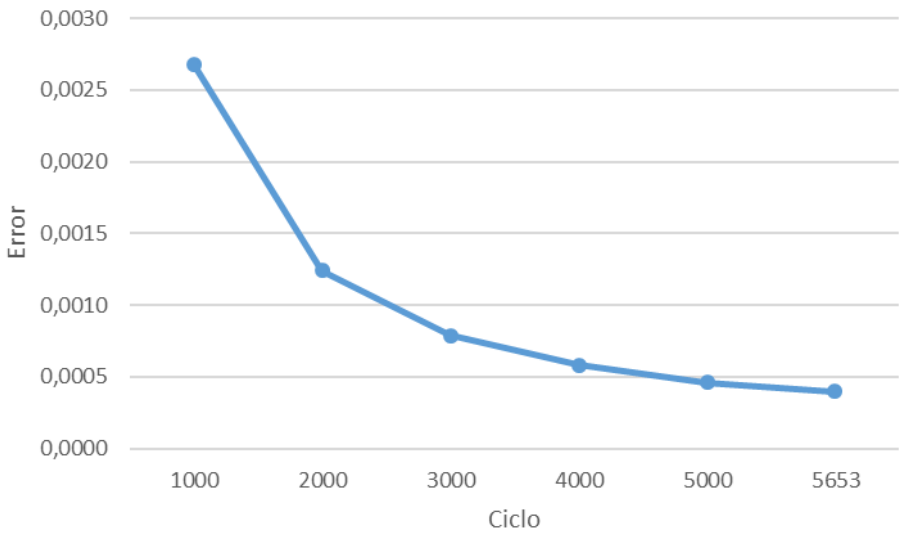

Figura 48 Evolución del error de la red neuronal por épocas.

Como se puede observar en la Figura 48 el error en el ciclo 1000 ya es muy pequeño, pero al requerirse un error mínimo de 0.0004, sigue iterando hasta alcanzarlo. Al alcanzar el

error mínimo definido, el entrenamiento finaliza, y con los valores de los pesos y bias definidos en la última iteración, se pasa a la fase de ejecución.

En la Figura 49 y la Figura 50 se muestran dos ejemplos de funcionamiento de la red neuronal, demostrándose que funciona correctamente.

```
11:48:10.178 -> Entrada 0: 1
11:48:23.432 -> Entrada 1: 1
11:48:24.766 -> Entrada 2: 1
11:48:25.689 -> Entrada 3: 1
11:48:25.689 -> Entrada 4: 1
11:48:27.089 -> Entrada 5: 1
11:48:28.390 -> Entrada 6: 1
11:48:28.390 -> Dato de entrada  1 1 1 1 1 1 1   Output 0.99118 0.00017 0.00172 0.00409
```

Figura 49 Prueba de la red neuronal (8).

```
11:49:20.411 -> Entrada 0: 1
11:49:37.851 -> Entrada 1: 1
11:49:38.741 -> Entrada 2: 1
11:49:39.493 -> Entrada 3: 0
11:49:40.280 -> Entrada 4: 0
11:49:41.408 -> Entrada 5: 1
11:49:42.638 -> Entrada 6: 1
11:49:43.392 -> Dato de entrada  1 1 1 0 0 1 1   Output 0.99342 0.00382 0.00003 0.99617
```

Figura 50 Prueba de la red neuronal (9).

6.2. Implementación en una Raspberry Pi y Tensorflow

En este apartado se implementa una red neuronal convolucional en una Raspberry Pi. Se ha escogido esta plataforma debido a su bajo coste, su capacidad de procesamiento y fácil programación mediante el uso de librerías. Los objetivos de esta implementación son los siguientes.

- Comprobar la implementabilidad para la resolución de un problema de complejidad medio-baja.
- Analizar la velocidad de procesamiento de datos de la plataforma.
- Analizar el tamaño de las redes implementables en una Raspberry Pi.

El software Tensorflow se ha utilizado para construir el modelo de red neuronal y entrenar la red neuronal, y Jupyter Notebook para poder ejecutar la red en modo predicción. Se ha tomado como base el código del artículo [67].

6.2.1. Problema a resolver

El problema que va a resolver la red neuronal será uno de reconocimiento de caracteres. Para ello, se va a utilizar la base de datos MNIST. Es el considerado "Hello World" de la visión artificial. Tiene un conjunto de 60.000 ejemplos de entrenamiento, y otro de 10.000

ejemplos de prueba. Los dígitos han sido normalizados en tamaño y centrados en una imagen de tamaño fijo. [68]

Se trata de una buena base de datos para probar técnicas de aprendizaje y métodos de reconocimiento de patrones con datos del mundo real, dedicando un esfuerzo mínimo al preprocesamiento y al formateo de los datos de entrenamiento.

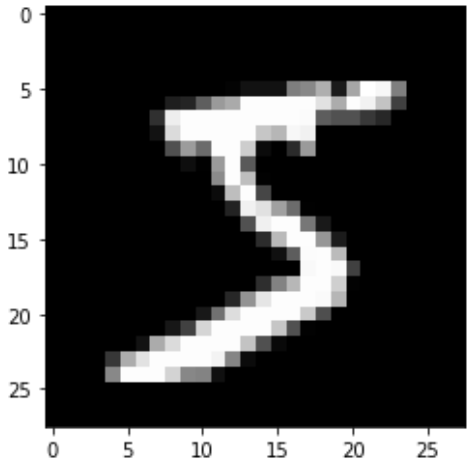

Figura 51 Muestra de la base de datos MNIST.

6.2.2. Red neuronal utilizada

Se utiliza una red neuronal convolucional, que consta de las siguientes capas:

- Entrada de imágenes de 28x28 píxeles.
- Capa convolucional 2D, con 64 kernel de 3x3, activación ReLu. Generan una salida de 26x26.
- Capa Maxpooling 2D, de 2x2. Convierte la entrada de 26x26 en 13x13.
- Capa convolucional 2D, con 32 kernel de 3x3 y activación relu. Convierten la entrada de 13x13 en 11x11.
- Capa Maxpooling 2D, de 2x2. Convierte la entrada de 11x11 a 5x5.
- Capa de aplanamiento.
- Capa totalmente interconectada, de 400 neuronas con la función de activación ReLu.
- Capa totalmente interconectada de 100 neuronas con la función de activación ReLu.
- Capa Dropout, que durante el entrenamiento introduce aleatoriamente en las entradas ceros con una frecuencia de 0.25 Hz, para ayudar a evitar el sobreajuste.
- Capa totalmente interconectada de 10 neuronas.
- Capa Softmax, con una función de activación softmax.

En la Tabla 12 se pueden observar la distribución de los parámetros entrenables que tiene la red neuronal.

Tabla 12 Parámetros entrenables de la red convolucional.

Capa	Parámetros
Conv2D_1	640
Conv2D_2	18.464
Capa totalmente interconectada 1	320.400
Capa totalmente interconectada 2	40.100
Capa totalmente interconectada 3	1.010
TOTAL	**380.614**

6.2.3. Montaje de la Raspberry Pi

Para realizar la implementación de la red neuronal en una Raspberry Pi, son necesarios los siguientes elementos:

- Raspberry Pi, cualquier modelo vale, pero el utilizado es la Raspberry Pi 4B.
- Pantalla, teclado y ratón para conectarse directamente, o un PC para conectarse vía remoto a la Raspberry Pi.

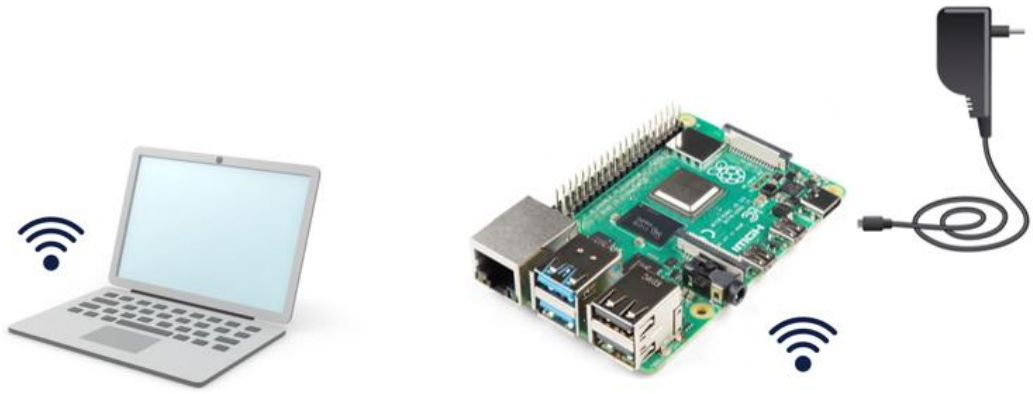

Figura 52 Montaje de la Raspberry Pi para programarla remotamente.

El sistema operativo Raspbian se descarga de la página oficial. [69] El sistema operativo se flashea a la tarjeta MicroSD mediante el PC con Win32DiskImager. [70] Es probable que la primera vez sea necesario conectarse directamente a la Raspberry para poder configurar la red wifi y cambiar la contraseña del usuario principal, pero si se hace vía ethernet, no es necesario.

Para conectarse remotamente, tras haber habilitado la interfaz SSH, se utiliza el programa PuTTY. [71]

6.2.4. Programación de la Raspberry Pi

Antes de instalar cualquier programa, es muy recomendable actualizar los paquetes de Linux a su versión más reciente. Esto actualiza las aplicaciones y software que estén en los repositorios oficiales.

```
sudo apt-get update

sudo apt-get upgrade
```

Para instalar TensorFlow, es necesario instalar Python, siendo compatible de las versiones 3.6 a 3.9.

```
sudo apt-get install python3 python3-dev python3-venv python3-pip libffi-dev libssl-dev
```

Ya se puede instalar TensorFlow.

```
pip install tensorflow
```

También es necesario instalar Jupyter Notebook, que es una herramienta que se va a utilizar para poder usar la red neuronal en modo de predicción después de haber sido entrenada.

```
pip3 install jupyterlab

pip3 install notebook
```

Para ejecutar Jupyter Notebook:

```
Jupyter notebook
```

Se abre una pestaña en el navegador, donde se puede crear un nuevo cuaderno e insertar el código de la red neuronal que se incluye en el ANEXO Nº1: CÓDIGO Y PROGRAMACIÓN.

También es probable que sea necesario instalar varios paquetes y librerías como numpy, h5py y matplotlib.pyplot.

Respecto al funcionamiento del código, sigue el siguiente proceso.

- Preparación de los datos de entrenamiento.

- Construcción del modelo de red neuronal.

- Entrenamiento. Se ejecuta en 10 épocas. En cada una, hay 100 imágenes de entrenamiento. Al finalizar cada época, se muestran los datos estadísticos del funcionamiento de la red neuronal: error, precisión, error con datos de validación y precisión con datos de validación.

6.2.5. Prueba de la red neuronal

6.2.5.1 Entrenamiento

Se ejecuta el código de la fase de entrenamiento. Es probable que aparezcan avisos en rojo, que no son importantes y se pueden ignorar.

```
Preparing Dataset...
Building Model...
Loading Data...
Training...
Epoch 1/10
100/100 - 49s - loss: 0.6606 - accuracy: 0.7965 - val_loss: 0.2139 -
val_accuracy: 0.9340

Epoch 00001: saving model to mnist_digits/
Epoch 2/10
100/100 - 27s - loss: 0.2004 - accuracy: 0.9405 - val_loss: 0.1097 -
val_accuracy: 0.9675
Epoch 00002: saving model to mnist_digits/
Epoch 3/10
100/100 - 32s - loss: 0.1317 - accuracy: 0.9610 - val_loss: 0.0897 -
val_accuracy: 0.9722

Epoch 00003: saving model to mnist_digits/
Epoch 4/10
100/100 - 29s - loss: 0.1095 - accuracy: 0.9668 - val_loss: 0.0575 -
val_accuracy: 0.9816

Epoch 00004: saving model to mnist_digits/
Epoch 5/10
100/100 - 28s - loss: 0.0933 - accuracy: 0.9726 - val_loss: 0.0567 -
val_accuracy: 0.9806

Epoch 00005: saving model to mnist_digits/
Epoch 6/10
100/100 - 23s - loss: 0.0719 - accuracy: 0.9775 - val_loss: 0.0641 -
val_accuracy: 0.9785

Epoch 00006: saving model to mnist_digits/
Epoch 7/10
100/100 - 22s - loss: 0.0780 - accuracy: 0.9768 - val_loss: 0.0507 -
val_accuracy: 0.9831

Epoch 00007: saving model to mnist_digits/
Epoch 8/10
```

```
100/100 - 24s - loss: 0.0638 - accuracy: 0.9818 - val_loss: 0.0394 -
val_accuracy: 0.9878

Epoch 00008: saving model to mnist_digits/
Epoch 9/10
100/100 - 22s - loss: 0.0638 - accuracy: 0.9814 - val_loss: 0.0406 -
val_accuracy: 0.9868

Epoch 00009: saving model to mnist_digits/
Epoch 10/10
100/100 - 29s - loss: 0.0500 - accuracy: 0.9839 - val_loss: 0.0379 -
val_accuracy: 0.9873

Epoch 00010: saving model to mnist_digits/
training took: 324.6543 secs.
```

Se puede observar como el entrenamiento de la red tarda alrededor de 5 minutos 24 segundos en entrenar la red neuronal, y como se puede observar en la Figura 53, usa el 100% de la CPU.

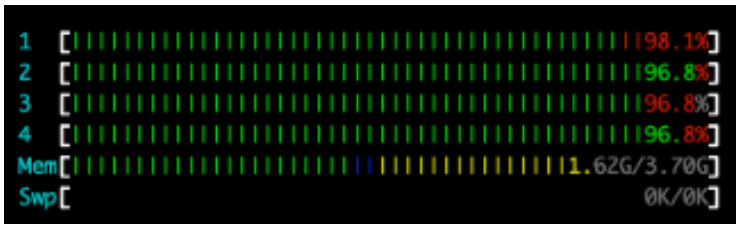

Figura 53 Estado de la Raspberry Pi durante el entrenamiento.

En la Figura 54 se puede observar cómo a partir de la época 6, la red mejora muy lentamente.

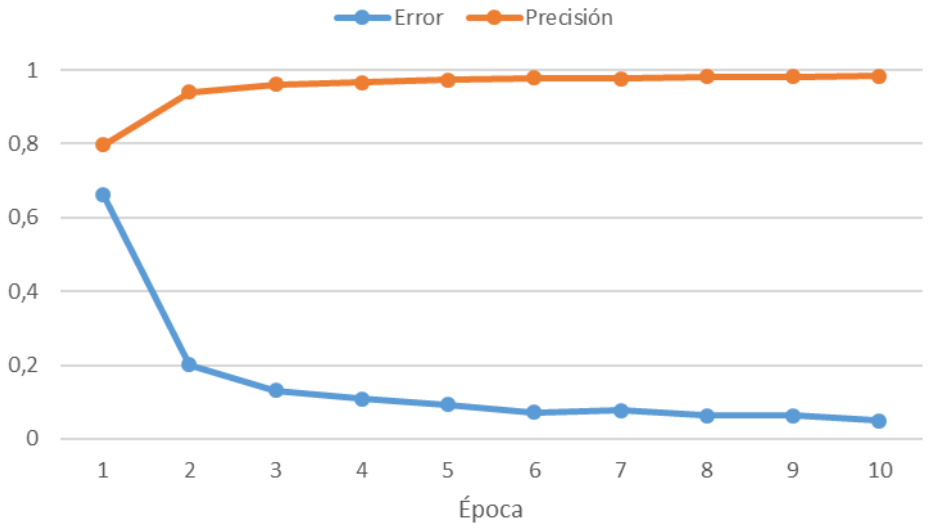

Figura 54 Error y precisión durante el entrenamiento.

6.2.5.2 Predicción

Si se ejecuta el código en modo predicción, se pueden introducir dígitos escritos a mano por el usuario. La red neuronal da una predicción, y un porcentaje de la confianza que tiene en la predicción realizada, como se puede ver en las pruebas realizadas en la Figura 55.

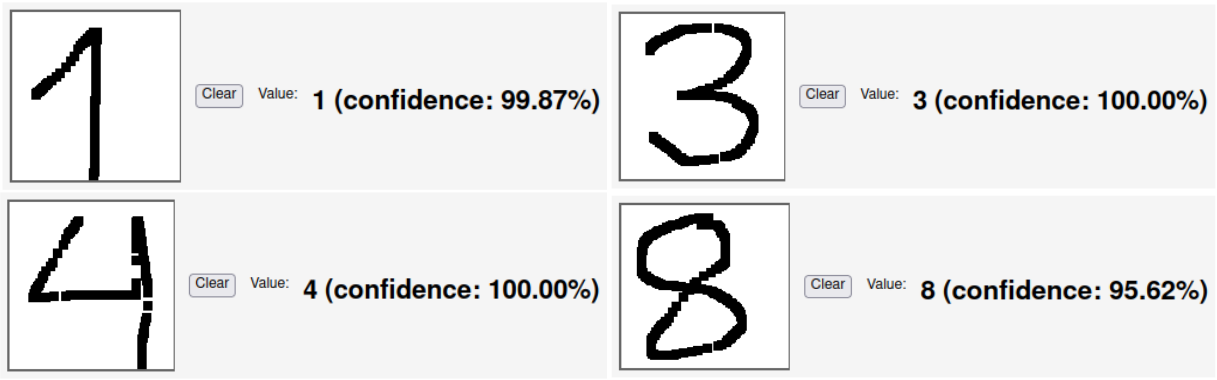

Figura 55 Prueba de la red neuronal con diferentes dígitos.

7. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURAS

Las redes neuronales, la inteligencia artificial, el deep learning… Son conceptos que se están desarrollando a una velocidad muy grande durante los últimos años, y que aumentan en complejidad y utilidad según se desarrolla la tecnología.

La capacidad de aprender de ejemplos, que sean tolerantes a errores y capaces de gestionar datos incompletos o ruidosos, y que puedan afrontar problemas no lineales convierten a las redes neuronales en candidatos idóneos para su aplicación en muchos ámbitos, y en este caso, también para aplicaciones de Ingeniería Eléctrica.

Tradicionalmente, la implementación de redes neuronales se ha realizado principalmente en software, quedando la implementación hardware en un segundo plano, donde destacaban las FPGA y los ASIC.

En los últimos años, gracias a los avances tecnológicos, han surgido nuevas plataformas que están preparadas para entrenar y ejecutar redes neuronales, como son Arduino, Raspberry Pi, Jetson Nano o Google Coral. Técnicamente, no son plataformas hardware, ya que los algoritmos de las redes neuronales se programan mediante código, pero al ser dispositivos de tamaño reducido, de gran potencia de procesamiento, son una opción muy interesante para la implementación de redes neuronales.

Entre todas son herramientas distintas, que destacan en cosas distintas. Ciertas tareas pueden realizarse con todas, pero en algunas resulta mucho más adecuado y eficiente emplear una de ellas. Aunque, a veces, es probable que una sola no sea la solución, y que se pueda ganar en versatilidad combinando varias de las opciones.

Las FPGA son una plataforma ideal para la implementación de redes neuronales, pero la complejidad de su programación y el poco soporte disponible en la red, junto con el alto coste de los kits de desarrollo hacen que sea una opción menos interesante.

Como se ha demostrado en este trabajo, la red neuronal más grande que se puede realizar con Arduino es de aproximadamente 20 neuronas, que es más bien una red pequeña. Otra limitación que tiene es que no se pueden introducir demasiados datos de entrenamiento, ya que la memoria que tiene es limitada. Esto se podría solucionar añadiendo un módulo al Arduino que le permita leer datos de una tarjeta micro SD.

El ejemplo realizado, incorpora en el mismo programa la fase de entrenamiento y de ejecución, para comprobar el funcionamiento de ambos procesos. Si se quiere implementar una red neuronal en un sistema real, haciendo uso de sus entradas y salidas, no interesa que, cada vez que se apague el Arduino haya que volver a entrenar la red. Sería interesante hacer un programa que entrene la red leyendo los datos de una tarjeta micro SD como se ha comentado, y que guarde las características de la red en un fichero. Después, habría que hacer otro programa enfocado a la fase de ejecución, que simplemente cargue en el inicio las características de la red neuronal, y se ejecute en bucle.

En conclusión, Arduino no es una de las plataformas más interesantes. Principalmente, por su reducida velocidad de procesamiento, memoria, y porque no es capaz de realizar procesamiento en paralelo (aunque en la comunidad de Arduino de internet parece que hay formas de hacer que se ejecute más de una tarea simultáneamente).

Debido a la gran cantidad de pines de entrada y salida y canales PWM, su uso sí que podría ser interesante como paso previo a una red neuronal implementada en otra plataforma (encargada de realizar el trabajo pesado de computación de datos), para realizar una gestión de los sensores. También podría funcionar como elemento posterior, para que una vez la red neuronal calcule las salidas, el Arduino genere las salidas necesarias. Implementar una red neuronal en Arduino sería viable para redes de pequeño tamaño y donde la velocidad de respuesta de la red no sea un requisito crítico.

La Raspberry Pi, en cambio, es un dispositivo mucho más versátil. Está diseñada para realizar proyectos que tienen una complejidad mayor, y unos requerimientos de computación mayores. Su compatibilidad con una gran cantidad de softwares potentes de diseño de redes neuronales convierte a la Raspberry Pi en una opción muy interesante. En el ejemplo realizado, se puede ver que se puede entrenar y ejecutar una red neuronal de un tamaño medio-bajo fácilmente, y con unos tiempos de entrenamiento asequibles.

En su esencia, la Raspberry Pi es una plataforma enfocada a que se pueda utilizar en muchos tipos de proyectos. El Jetson Nano y el Google Coral, en cambio, están específicamente enfocados al desarrollo de redes neuronales, por lo que el rendimiento de estos es muy superior al de la Raspberry Pi a la hora de entrenar redes neuronales. Para redes neuronales más pesadas, en cambio, se puede ver en los estudios de comparación con Jetson Nano y Google Coral que el rendimiento de la Raspberry Pi es muy bajo, y que no es suficiente para la implementación de estas redes.

Figura 56 Clúster de 4 Raspberry Pi.

Como solución, se pueden plantear soluciones como añadir uno o más de los aceleradores USB mencionados en las comparaciones, que se encargan de mejorar mucho el rendimiento de la Raspberry Pi. Otra opción sería hacer un clúster de varias Raspberry Pi, como se muestra en la Figura 56. De esta forma, aumentaría tanto la capacidad de procesamiento como la memoria RAM, por lo que sería una buena opción para implementar redes neuronales más grandes.

Finalmente, sería interesante aplicar un problema del ámbito de la ingeniería eléctrica a una de las soluciones presentadas, y comprobar su viabilidad técnica y económica sobre los sistemas actuales.

8. BIBLIOGRAFÍA

[1] J. M. F. F. Raquel Flórez López, Las Redes Neuronales Artificiales.

[2] I. S. Janardan Misra, Artificial neuralnetworks in hardware: A survey of two decades of progress.

[3] V. V. Santiago, Aplicación de las redes neuronales artificiales para la detección en tiempo real del fenómeno de la ferrorresonancia en transformadores de tensión.

[4] W. T. Illingworth, Beginners Guide to Neural Networks.

[5] M. M. Mijwel, Artificial Neural Networks Advantages and Disadvantages.

[6] A. M. Munt, Introducción a los modelos de redes neuronales artificiales. El perceptrón simple y multicapa.

[7] «Hopfield network,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Hopfield_network.

[8] «Neurona de McCulloch-Pitts,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Neurona_de_McCulloch-Pitts.

[9] X. B. Olabe, Redes Neuronales Artificiales y sus aplicaciones.

[10] «Radial Base Function,» Science Direct, [En línea]. Available: https://www.sciencedirect.com/topics/engineering/radial-base-function.

[11] «Radial Basis Function Network,» ScienceDirect, [En línea]. Available: https://www.sciencedirect.com/topics/engineering/radial-basis-function-network.

[12] «Kohonen Self-Organizing Maps,» Towards Data Science, [En línea]. Available: https://towardsdatascience.com/kohonen-self-organizing-maps-a29040d688da.

[13] «Redes Autoorganizadas. Redes SOFM,» [En línea]. Available: https://www.ibiblio.org/pub/linux/docs/LuCaS/Presentaciones/200304curso-glisa/redes_neuronales/curso-glisa-redes_neuronales-html/x152.html.

[14] «Mapa autoorganizado,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Mapa_autoorganizado.

[15] G. D. H. Robert DiPietro, «Chapter 21 - Deep learning: RNNs and LSTM,» Science Direct, [En línea]. Available: https://www.sciencedirect.com/science/article/pii/B9780128161760000260.

[16] J. Torres, «Redes Neuronales Recurrentes,» Jordi TORRES.AI, [En línea]. Available: https://torres.ai/redes-neuronales-recurrentes/.

[17] «A Guide to RNN: Understanding Recurrent Neural Networks and LSTM Networks,» Builtin, [En línea]. Available: https://builtin.com/data-science/recurrent-neural-networks-and-lstm.

[18] «Hopfield Neural Network,» GeeksforGeeks, [En línea]. Available: https://www.geeksforgeeks.org/hopfield-neural-network/.

[19] S. Saha, «A Comprehensive Guide to Convolutional Neural Networks,» Towards Data Science, [En línea]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[20] N. Developer, «Artificial Neural Network,» [En línea]. Available: https://developer.nvidia.com/discover/artificial-neural-network.

[21] P. S. P. W. R. S. Marcin Korytkowski, Fast Computing Framework for Convolutional Neural Networks.

[22] «Convolutional neural network,» Wikipedia, [En línea]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network.

[23] S. Kalogirou, Applications of artificial neural networks in energy systems. A review.

[24] F. F. D. B. R. V. F. M.-D. Sandy Abreu, A survey of software and hardware use in Artificial Neural Networks.

[25] A. K. M. Tarafdar Haque, Application of Neural Networks in Power Systems; A Review.

[26] N. M. S. B. M. S. A. Wafi Danesh, A Review of Neural Networking Methodology to Different Aspects of Electrical Power Systems.

[27] P. D. Kothari, Application of neural networks to power systems.

[28] R. C. Bansal, Overview and Literature Survey of Artificial Neural Networks Applications to Power Systems (1992-2004).

[29] M. J. S. M. S. M. A. M. R. K. Mostafa Gholami, Static security assessment of power systems: A review.

[30] C. K. B. V. Tikhonov E.E., Hardware and Software Implementation of Neural Network Control of Power Systems based on the System of Residual Classes.

[31] «MATLAB,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/MATLAB.

[32] «Redes neuronales,» MathWorks, [En línea]. Available: https://es.mathworks.com/discovery/neural-network.html.

[33] «Deep Learning Toolbox,» MathWorks, [En línea]. Available: https://es.mathworks.com/products/deep-learning.html.

[34] «Por qué TensorFlow,» TensorFlow, [En línea]. Available: https://www.tensorflow.org/?hl=es-419.

[35] «TensorFlow,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/TensorFlow.

[36] «TensorFlow,» Github, [En línea]. Available: https://github.com/tensorflow.

[37] «Getting started with Tensorflow 2.0,» Data Driven Investor, [En línea]. Available: https://medium.datadriveninvestor.com/getting-started-with-tensorflow-2-0-53d4e9d04c57.

[38] «PyTorch: end-to-end machine learning framework,» PyTorch, [En línea]. Available: https://pytorch.org/features/.

[39] «PyTorch,» Github, [En línea]. Available: https://github.com/pytorch/pytorch.

[40] «PyTorch,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/PyTorch.

[41] «¿Qué PC debo comprar para deep learning?,» Modelizame, [En línea]. Available: https://modeliza.me/blog/que-pc-debo-comprar-para-deep-learning/.

[42] J. Dsouza, «What is a GPU and do you need one in Deep Learning?,» Towards Data Science. [En línea].

[43] Y. Fernández, «Qué es Arduino, cómo funciona y qué puedes hacer con uno,» Xataka, [En línea]. Available: https://www.xataka.com/basics/que-arduino-como-funciona-que-puedes-hacer-uno.

[44] «Arduino UNO,» Arduino, [En línea]. Available: https://arduino.cl/arduino-uno/.

[45] «46 proyectos makers para hacer en verano con Arduino y Raspberry Pi,» Xataka, [En línea]. Available: https://www.xataka.com/makers/46-proyectos-makers-para-hacer-verano-arduino-raspberry-pi.

[46] «Raspberry Pi,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Raspberry_Pi.

[47] «Características de la nueva Raspberry Pi 4 Model B+,» PC Componentes, [En línea]. Available: https://www.pccomponentes.com/caracteristicas-raspberry-pi-4.

[48] «25 proyectos con Raspberry Pi que explotan todo su potencial,» Ionos Digital Guide, [En línea]. Available: https://www.ionos.es/digitalguide/servidores/know-how/un-vistazo-a-proyectos-basados-en-raspberry-pi/.

[49] «Jetson Nano Developer Kit,» NVIDIA Developer, [En línea]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[50] «Nvidia lanza Jetson Nano, un Mini PC de 109 € para fans de la robótica y la inteligencia artificial,» Xataka, [En línea]. Available: https://www.xataka.com/inteligencia-artificial/nvidia-lanza-jetson-nano-mini-pc-109-eur-para-fans-robotica-inteligencia-artificial.

[51] «Google lanza al mercado Coral, la placa de 150 dólares para desarrolladores de inteligencia artificial en edge computing,» Xataka, [En línea]. Available: https://www.xataka.com/inteligencia-artificial/google-lanza-al-mercado-coral-placa-150-dolares-para-desarrolladores-inteligencia-artificial-edge-computing.

[52] «Circuito integrado de aplicación específica,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Circuito_integrado_de_aplicaci%C3%B3n_espec%C3%ADfica.

[53] «CPLD,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/CPLD.

[54] «Así son los CPLD, la alternativa de menor coste a los FPGA,» Hardzone, [En línea]. Available: https://hardzone.es/reportajes/que-es/cpld/.

[55] «Field-programmable gate array,» Wikipedia, [En línea]. Available: https://es.wikipedia.org/wiki/Field-programmable_gate_array.

[56] «Qué es un FPGA: características y utilidad de este tipo de componente,» HardZone, [En línea]. Available: https://hardzone.es/reportajes/que-es/fpga-caracteristicas-utilidad/.

[57] D. Young, A. Lee y L. Cheng, Hardware realisation of artificial neural network with application to information coding.

[58] «Arduino MKR Vidor 4000,» Arduino Store, [En línea]. Available: https://store.arduino.cc/products/arduino-mkr-vidor-4000.

[59] «Papilio Wiki,» Gadget Factory, [En línea]. Available: https://papilio.cc/index.php?n=Papilio.Papilio.

[60] «¿QUÉ ES UNA FPGA? MOTIVOS DE SU AUGE EN LA COMUNIDAD MAKER,» Luis Llamas, [En línea]. Available: https://www.luisllamas.es/que-es-una-fpga/.

[61] «Deep learning with FPGA aka Binary Neural Networks,» Q-engineering, [En línea]. Available: https://qengineering.eu/deep-learning-with-fpga-aka-bnn.html.

[62] «Raspberry Pi vs Arduino, ¿en qué se diferencian y para qué se usan?,» Hard Zone, [En línea]. Available: https://hardzone.es/reportajes/comparativas/raspberry-pi-vs-arduino/.

[63] «Deep learning with Raspberry Pi and alternatives in 2021,» Q-engineering, [En línea]. Available: https://qengineering.eu/deep-learning-with-raspberry-pi-and-alternatives.html.

[64] «Is Google Coral worthy to buy? Better than Rasp Pi 4 or Jetson Nano?,» Artificial, [En línea]. Available: https://medium.com/@deve321/is-google-coral-worthy-to-buy-better-than-rasp-pi-4-or-jetson-nano-819cac61a537.

[65] «Arduino Software,» Arduino, [En línea]. Available: https://www.arduino.cc/en/software.

[66] R. Heymsfeld, «A neural network for arduino,» Hobbizine, [En línea]. Available: http://robotics.hobbizine.com/arduinoann.html.

[67] «Running a Convolutional Neural Network on Raspberry PI,» Start It Up, [En línea]. Available: https://medium.com/swlh/running-a-convolutional-neural-network-on-raspberry-pi-4fc5bd80aa4d.

[68] C. C. C. J. B. Yann LeCun, «THE MNIST DATABASE OF HANDWRITTEN DIGITS,» [En línea]. Available: http://yann.lecun.com/exdb/mnist/.

[69] «Operating system images,» Raspberry Pi Foundation, [En línea]. Available: https://www.raspberrypi.org/software/operating-systems/.

[70] «Win32 Disk Imager,» Source Forge, [En línea]. Available: https://sourceforge.net/projects/win32diskimager/.

[71] «Download PuTTY,» PuTTY, [En línea]. Available: https://www.putty.org/.

BIZKAIAKO INGENIARITZA ESKOLA

ESCUELA DE INGENIERÍA DE BILBAO

ANEXO N°1: CÓDIGO Y PROGRAMACIÓN

MASTER EN INTEGRACIÓN DE LAS ENERGÍAS RENOVABLES EN EL SISTEMA ELÉCTRICO.

AUTOR: XABIER MAESTRE BETOLAZA

TUTORES: JULEN GOMEZ-CORNEJO, VICTOR VALVERDE

CURSO: 2020/2021

1. RNA ARDUINO

```
#include <math.h>

/******************************************************************
 * Network Configuration - customized per network
 ******************************************************************/

const int PatternCount = 10;
const int InputNodes = 7;
const int HiddenNodes = 8;
const int OutputNodes = 4;
const float LearningRate = 0.3;
const float Momentum = 0.9;
const float InitialWeightMax = 0.5;
const float Success = 0.0004;

const byte Input[PatternCount][InputNodes] = {
  { 1, 1, 1, 1, 1, 1, 0 },  // 0
  { 0, 1, 1, 0, 0, 0, 0 },  // 1
  { 1, 1, 0, 1, 1, 0, 1 },  // 2
  { 1, 1, 1, 1, 0, 0, 1 },  // 3
  { 0, 1, 1, 0, 0, 1, 1 },  // 4
  { 1, 0, 1, 1, 0, 1, 1 },  // 5
  { 0, 0, 1, 1, 1, 1, 1 },  // 6
  { 1, 1, 1, 0, 0, 0, 0 },  // 7
  { 1, 1, 1, 1, 1, 1, 1 },  // 8
  { 1, 1, 1, 0, 0, 1, 1 }   // 9
};

const byte Target[PatternCount][OutputNodes] = {
  { 0, 0, 0, 0 },
  { 0, 0, 0, 1 },
  { 0, 0, 1, 0 },
  { 0, 0, 1, 1 },
  { 0, 1, 0, 0 },
  { 0, 1, 0, 1 },
  { 0, 1, 1, 0 },
  { 0, 1, 1, 1 },
  { 1, 0, 0, 0 },
  { 1, 0, 0, 1 }
};

/******************************************************************
 * End Network Configuration
 ******************************************************************/


int i, j, p, q, r;
int funcion;
int ReportEvery1000;
int RandomizedIndex[PatternCount];
long  TrainingCycle;
float Rando;
```

```
float Error;
float Accum;


float Hidden[HiddenNodes];
float Output[OutputNodes];
float HiddenWeights[InputNodes+1][HiddenNodes];
float OutputWeights[HiddenNodes+1][OutputNodes];
float HiddenDelta[HiddenNodes];
float OutputDelta[OutputNodes];
float ChangeHiddenWeights[InputNodes+1][HiddenNodes];
float ChangeOutputWeights[HiddenNodes+1][OutputNodes];
const byte numChars = 32;
char receivedChars[numChars];   // an array to store the received data

boolean newData = false;

int dataNumber = 0;             // new for this version

void setup() {
    Serial.begin(9600);
    Serial.println("<Arduino is ready>");

    Serial.println ("Elige el tipo de función de transferencia");
    Serial.println("1.- Lineal\n2.- Sigmoidea");

    recvWithEndMarker();
    showNewNumber();

    funcion = dataNumber;

    Serial.print ("Función de transferencia elegida  ");
    switch(funcion){
      case 1:
        Serial.println ("1.- Lineal");
        break;
      case 2:
        Serial.println ("2.- Sigmoidea");
        break;
    }
    Serial.print("\n\n");

    randomSeed(analogRead(3));
    ReportEvery1000 = 1;
    for( p = 0 ; p < PatternCount ; p++ ) {
      RandomizedIndex[p] = p ;
    }



/******************************************************************
* Initialize HiddenWeights and ChangeHiddenWeights
******************************************************************/

  for( i = 0 ; i < HiddenNodes ; i++ ) {
    for( j = 0 ; j <= InputNodes ; j++ ) {
      ChangeHiddenWeights[j][i] = 0.0 ;
      Rando = float(random(100))/100;
      HiddenWeights[j][i] = 2.0 * ( Rando - 0.5 ) * InitialWeightMax ;
```

```
    }
  }
/*************************************************************
* Initialize OutputWeights and ChangeOutputWeights
*************************************************************/

  for( i = 0 ; i < OutputNodes ; i ++ ) {
    for( j = 0 ; j <= HiddenNodes ; j++ ) {
      ChangeOutputWeights[j][i] = 0.0 ;
      Rando = float(random(100))/100;
      OutputWeights[j][i] = 2.0 * ( Rando - 0.5 ) * InitialWeightMax ;
    }
  }
  Serial.println("Initial/Untrained Outputs: ");
  toTerminal();
/*************************************************************
* Begin training
*************************************************************/

  for( TrainingCycle = 1 ; TrainingCycle < 2147483647 ; TrainingCycle++) {

/*************************************************************
* Randomize order of training patterns
*************************************************************/

    for( p = 0 ; p < PatternCount ; p++) {
      q = random(PatternCount);
      r = RandomizedIndex[p] ;
      RandomizedIndex[p] = RandomizedIndex[q] ;
      RandomizedIndex[q] = r ;
    }
    Error = 0.0 ;
/*************************************************************
* Cycle through each training pattern in the randomized order
*************************************************************/
    for( q = 0 ; q < PatternCount ; q++ ) {
      p = RandomizedIndex[q];

/*************************************************************
* Compute hidden layer activations
*************************************************************/

      for( i = 0 ; i < HiddenNodes ; i++ ) {
        Accum = HiddenWeights[InputNodes][i] ;
        for( j = 0 ; j < InputNodes ; j++ ) {
          Accum += Input[p][j] * HiddenWeights[j][i] ;
        }
        Hidden[i] = ftransf (Accum,funcion) ;
      }

/*************************************************************
* Compute output layer activations and calculate errors
*************************************************************/

      for( i = 0 ; i < OutputNodes ; i++ ) {
        Accum = OutputWeights[HiddenNodes][i] ;
        for( j = 0 ; j < HiddenNodes ; j++ ) {
          Accum += Hidden[j] * OutputWeights[j][i] ;
        }
```

```
        Output[i] = ftransf (Accum,funcion) ;
        OutputDelta[i] = (Target[p][i] - Output[i]) * Output[i] * (1.0 -
Output[i]) ;
        Error += 0.5 * (Target[p][i] - Output[i]) * (Target[p][i] -
Output[i]) ;
      }

/******************************************************************
* Backpropagate errors to hidden layer
******************************************************************/

      for( i = 0 ; i < HiddenNodes ; i++ ) {
        Accum = 0.0 ;
        for( j = 0 ; j < OutputNodes ; j++ ) {
          Accum += OutputWeights[i][j] * OutputDelta[j] ;
        }
        HiddenDelta[i] = Accum * Hidden[i] * (1.0 - Hidden[i]) ;
      }


/******************************************************************
* Update Inner-->Hidden Weights
******************************************************************/


      for( i = 0 ; i < HiddenNodes ; i++ ) {
        ChangeHiddenWeights[InputNodes][i] = LearningRate * HiddenDelta[i]
+ Momentum * ChangeHiddenWeights[InputNodes][i] ;
        HiddenWeights[InputNodes][i] += ChangeHiddenWeights[InputNodes][i]
;
        for( j = 0 ; j < InputNodes ; j++ ) {
          ChangeHiddenWeights[j][i] = LearningRate * Input[p][j] *
HiddenDelta[i] + Momentum * ChangeHiddenWeights[j][i];
          HiddenWeights[j][i] += ChangeHiddenWeights[j][i] ;
        }
      }

/******************************************************************
* Update Hidden-->Output Weights
******************************************************************/

      for( i = 0 ; i < OutputNodes ; i ++ ) {
        ChangeOutputWeights[HiddenNodes][i] = LearningRate * OutputDelta[i]
+ Momentum * ChangeOutputWeights[HiddenNodes][i] ;
        OutputWeights[HiddenNodes][i] +=
ChangeOutputWeights[HiddenNodes][i] ;
        for( j = 0 ; j < HiddenNodes ; j++ ) {
          ChangeOutputWeights[j][i] = LearningRate * Hidden[j] *
OutputDelta[i] + Momentum * ChangeOutputWeights[j][i] ;
          OutputWeights[j][i] += ChangeOutputWeights[j][i] ;
        }
      }
    }

/******************************************************************
* Every 1000 cycles send data to terminal for display
******************************************************************/
    ReportEvery1000 = ReportEvery1000 - 1;
    if (ReportEvery1000 == 0)
```

```
    {
      Serial.println();
      Serial.println();
      Serial.print ("TrainingCycle: ");
      Serial.print (TrainingCycle);
      Serial.print ("  Error = ");
      Serial.println (Error, 5);

      toTerminal();

      if (TrainingCycle==1)
      {
        ReportEvery1000 = 999;
      }
      else
      {
        ReportEvery1000 = 1000;
      }
    }


/*******************************************************************
* If error rate is less than pre-determined threshold then end
*******************************************************************/

    if( Error < Success ) break ;
  }
  Serial.println ();
  Serial.println();
  Serial.print ("TrainingCycle: ");
  Serial.print (TrainingCycle);
  Serial.print ("  Error = ");
  Serial.println (Error, 5);

  toTerminal();

  Serial.println ();
  Serial.println ();
  Serial.println ("Training Set Solved! ");
  ReportEvery1000 = 1;

}


/*******************************************************************
* Fase de ejecución
*******************************************************************/

void loop() {

    int arrayLen = 7;
    int Entrada[arrayLen];

    Serial.println ("\n--------\n");
    Serial.println ("Introduce valores de 0 a 1");
    Serial.println ();
    for(i=0;i<arrayLen;i++){

      Serial.print("Entrada ");
```

```
      Serial.print(i);
      Serial.print(": ");
      recvWithEndMarker();
      showNewNumber();
      Entrada[i]= dataNumber;
      Serial.println(Entrada[i]);

      while (Serial.available() > 0) {
      Serial.print("delay");
        //delay(200);
        Serial.read();
      }
    }

    for( i = 0 ; i < HiddenNodes ; i++ ) {
      Accum = HiddenWeights[InputNodes][i] ;
      for( j = 0 ; j < InputNodes ; j++ ) {
        Accum += Entrada[j] * HiddenWeights[j][i] ;
      }
      Hidden[i] = ftransf(Accum,funcion) ;
    }

    for( i = 0 ; i < OutputNodes ; i++ ) {
      Accum = OutputWeights[HiddenNodes][i] ;
      for( j = 0 ; j < HiddenNodes ; j++ ) {
        Accum += Hidden[j] * OutputWeights[j][i] ;
      }
      Output[i] = ftransf(Accum,funcion) ;
    }
    Serial.print ("Dato de entrada  ");
    for( i = 0 ; i < InputNodes ; i++ ) {
      Serial.print (Entrada[i],5);
      Serial.print (" ");
    }

    Serial.print ("  Output ");
    for( i = 0 ; i < OutputNodes ; i++ ) {
      Serial.print (Output[i],5);
      Serial.print (" ");
    }
    Serial.print("\n");
}


/******************************************************************
* Funciones definidas por el usuario
******************************************************************/


void recvWithEndMarker() {
    static byte ndx = 0;
    char endMarker = '\n';
    char rc;

    while(Serial.available() <= 0){
        delay(1000);
    }

    while (Serial.available() > 0) {
```

```
        rc = Serial.read();

        if(rc != endMarker) {
            receivedChars[ndx] = rc;
            ndx++;
            if (ndx >= numChars) {
                ndx = numChars - 1;
            }
        }
        else {
            receivedChars[ndx] = '\0'; // terminate the string
            ndx = 0;
            newData = true;
        }
    }
}

void showNewNumber() {

    if (newData == true) {
        dataNumber = 0;             // new for this version
        dataNumber = atoi(receivedChars);   // new for this version
        newData = false;
    }
    else{
      Serial.print("false");
    }
}

float ftransf(float x, int funcion1){
  if (funcion1 == 1){
      return x;
  }
  if (funcion1 == 2) {
  float y;
  y = 1.0/(1.0 + exp(-x));
  return y;
  }
}

void toTerminal()
{
  for( p = 0 ; p < PatternCount ; p++ ) {
    Serial.println();
    Serial.print ("  Training Pattern: ");
    Serial.println (p);
    Serial.print ("  Input ");
    for( i = 0 ; i < InputNodes ; i++ ) {
      Serial.print (Input[p][i], DEC);
      Serial.print (" ");
    }
    Serial.print ("  Target ");
    for( i = 0 ; i < OutputNodes ; i++ ) {
      Serial.print (Target[p][i], DEC);
      Serial.print (" ");
    }

/******************************************************************
* Compute hidden layer activations
```

```
******************************************************************/

    for( i = 0 ; i < HiddenNodes ; i++ ) {
      Accum = HiddenWeights[InputNodes][i] ;
      for( j = 0 ; j < InputNodes ; j++ ) {
        Accum += Input[p][j] * HiddenWeights[j][i] ;
      }
      Hidden[i] = ftransf (Accum,funcion) ;
    }

/******************************************************************
* Compute output layer activations and calculate errors
******************************************************************/

    for( i = 0 ; i < OutputNodes ; i++ ) {
      Accum = OutputWeights[HiddenNodes][i] ;
      for( j = 0 ; j < HiddenNodes ; j++ ) {
        Accum += Hidden[j] * OutputWeights[j][i] ;
      }
      Output[i] = ftransf (Accum,funcion) ;
    }
    Serial.print ("  Output ");
    for( i = 0 ; i < OutputNodes ; i++ ) {
      Serial.print (Output[i], 5);
      Serial.print (" ");
    }
  }

}
```

2. RNA RASPBERRY PI

2.1. Fase de entrenamiento

```
#!/usr/bin/env python
# coding: utf-8

import os
import shutil
import time

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

from tensorflow.keras import Sequential
from tensorflow.keras.callbacks import ModelCheckpoint, TensorBoard
from tensorflow.keras.layers import Dense, Flatten, Softmax, Conv2D,
Dropout, MaxPooling2D


print("Preparing Dataset...")
# Download dataset from internet (if not present)
mnist = tf.keras.datasets.mnist.load_data()
(x_train, y_train), (x_test, y_test) = mnist

# CONSTANTS
HEIGHT, WIDTH = x_train[0].shape
NCLASSES = tf.size(tf.unique(y_train).y)
BUFFER_SIZE = 5000
BATCH_SIZE = 100
NUM_EPOCHS = 10
STEPS_PER_EPOCH = 100


def scale(image, label):
    """scale pixel value from 0~255 to 0~1"""
    image = tf.cast(image, tf.float32)
    image /= 255
    image = tf.expand_dims(image, -1)
    return image, label

def load_dataset(training=True):
    """Loads MNIST dataset into a tf.data.Dataset"""
    (x_train, y_train), (x_test, y_test) = mnist
    x = x_train if training else x_test
    y = y_train if training else y_test
    # One-hot encode the classes
    y = tf.keras.utils.to_categorical(y, NCLASSES)
    dataset = tf.data.Dataset.from_tensor_slices((x, y))
    dataset = dataset.map(scale).batch(BATCH_SIZE)
    if training:
        dataset = dataset.shuffle(BUFFER_SIZE).repeat()
```

```
    return dataset

print("Building Model...")
# configure CNN
model = Sequential([
        Conv2D(64, kernel_size=3,
                activation='relu', input_shape=(WIDTH, HEIGHT, 1)),
        MaxPooling2D(2),
        Conv2D(32, kernel_size=3,
                activation='relu'),
        MaxPooling2D(2),
        Flatten(),
        Dense(400, activation='relu'),
        Dense(100, activation='relu'),
        Dropout(.25),
        Dense(10),
        Softmax()
    ])

model.compile(optimizer='adam',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

print("Loading Data...")
# Load train and validation datasets
train_data = load_dataset()
validation_data = load_dataset(training=False)

# save model and tensorboard data
OUTDIR = "mnist_digits/"
checkpoint_callback = ModelCheckpoint(
    OUTDIR, save_weights_only=True, verbose=1)
tensorboard_callback = TensorBoard(log_dir=OUTDIR)

# Train
print("Training...")
t1 = time.perf_counter()
history = model.fit(
    train_data,
    validation_data=validation_data,
    epochs=NUM_EPOCHS,
    steps_per_epoch=STEPS_PER_EPOCH,
    verbose=2,
    callbacks=[checkpoint_callback, tensorboard_callback]
)
t2 = time.perf_counter()
print("training took: {:4.4f} secs.".format(t2 - t1))
```

2.2. Fase de predicción

```
input_form = """
<table>
<td style="border-style: none;">
<div style="border: solid 2px #666; width: 143px; height: 144px;">
<canvas width="140" height="140"></canvas>
</div></td>
<td style="border-style: none;">
<button onclick="clear_value()">Clear</button>
```

```
</td>
<td>
Value:
</td>
<td>
<h1><span id="predicted">-</span></h1>
</td>
</table>
"""

javascript = '''
<script type="text/Javascript">
    var pixels = [];
    for (var i = 0; i < 28*28; i++) pixels[i] = 0;
    var click = 0;
    var canvas = document.querySelector("canvas");
    canvas.addEventListener("mousemove", function(e){
        if (e.buttons == 1) {
            click = 1;
            predicted.textContent = "-"
            canvas.getContext("2d").fillStyle = "rgb(0,0,0)";
            canvas.getContext("2d").fillRect(e.offsetX, e.offsetY, 8, 8);
            x = Math.floor(e.offsetY * 0.2);
            y = Math.floor(e.offsetX * 0.2) + 1;
            for (var dy = 0; dy < 2; dy++){
                for (var dx = 0; dx < 2; dx++){
                    if ((x + dx < 28) && (y + dy < 28)){
                        pixels[(y+dy)+(x+dx)*28] = 1;
                    }
                }
            }
        } else {
            if (click == 1) set_value();
            click = 0;
        }
    });

    var predicted = document.querySelector("#predicted");

    function set_value(){
        predicted.textContent = ". . ."
        var result = ""
        for (var i = 0; i < 28*28; i++) result += pixels[i] + ","
        var kernel = IPython.notebook.kernel;
        kernel.execute("pred = np.array(["+result+"]).reshape(HEIGHT,
WIDTH)");
        kernel.execute("pred = tf.cast(pred, tf.float32)");
        kernel.execute("pred = tf.expand_dims([pred], -1)");
        kernel.execute("pred = model.predict(pred)");
        kernel.execute("pred =  '{} (confidence:
{:02.2f}%)'.format(np.argmax(pred), np.max(pred)*100)");
        var callbacks = {
            iopub: {
                output: (data) => {
                    predicted.textContent = data.content.text.trim();
                }
            }
        };
        kernel.execute("print(pred)", callbacks);
```

```
    }

    function clear_value(){
        canvas.getContext("2d").fillStyle = "rgb(255,255,255)";
        canvas.getContext("2d").fillRect(0, 0, 140, 140);
        for (var i = 0; i < 28*28; i++) pixels[i] = 0;
        predicted.textContent = "-"
    }
</script>
'''

from IPython.display import HTML
HTML(input_form + javascript)
```