Universidad Zaragoza

Facultad de Ciencias
Universidad Zaragoza

Máster Universitario en

Física y Tecnologías Físicas

2019-2020

*Trabajo Fin de Máster*

# Aplicación de sistemas digitales programables en la extracción de información por fusión sensorial en dispositivos ubicuos


Daniel Enériz Orta

Tutores

Dr. Nicolás Medrano Marqués

Dra. Belén Calvo López


Zaragoza, 10 de julio de 2020

*Gracias a Nicolás y Belén por la oportunidad de enfrentarme a este problema y por su ayuda y libertad durante todo el proceso.*

# Índice general

# Resumen

Debido a su versatilidad, las redes neuronales artificiales (*Artificial Neural Networks, ANNs* [1]) han demostrado su eficacia en resolver y modelar problemas complejos en infinidad de aplicaciones dentro de los campos englobados en lo que se conoce como *inteligencia artificial*: procesamiento de lenguaje natural (*Natural Language Processing, NLP*), visión artificial, *big data*…Además de todas estas aplicaciones tan en auge, hay otras aplicaciones en el ámbito del procesamiento de medidas físico-químicas en las que las ANN son igualmente interesantes por su flexibilidad, fácil reprogramabilidad y modularidad de su diseño. Algunos ejemplos son: la linealización de sensores, fusión sensorial y la detección de VOCs (*Volatile Organic Compounds*) en aire...

En este trabajo se presenta un estudio sobre la aplicación de las redes neuronales en la adquisición, procesado de datos y extracción de información sin necesidad de ser específicamente integradas en silicio, ni tener que ser virtualizadas mediante software sobre grandes computadores. Para ello, consideramos su implementación sobre matrices de puertas lógicas programables (FPGAs, *Field-Programmable Gate Arrays*). En cualquier curso de Sistemas Digitales, las primeras entidades que se presentan como unidades de procesamiento son las puertas lógicas que, siguiendo el álgebra de Boole, permiten realizar cualquier función lógica y, por ende, adecuadamente conectadas, cualquier operación. De aquí nace la idea de una FPGA, consistente en circuitos integrados compuestos por matrices de bloques lógicos básicos (CLBs, *Configurable Logic Blocks*), cuya operación e interconexión puede ser fijada físicamente para realizar las operaciones requeridas de una manera óptima, permitiendo procesar información aportando una nueva característica frente a un computador convencional, en el que la mayoría de las operaciones y cálculos los ejecuta su unidad central de proceso o CPU (*Central Processing Unit*) o, en sistemas más avanzados y específicos, su unidad gráfica de procesado (GPU), a través de un código o secuencia de operaciones que se realiza consecutivamente, o con un bajo nivel de paralelización: la concurrencia.

Es precisamente esta característica de paralelización la que las hace especialmente interesantes como plataformas sobre las que implementar redes neuronales, cuyo procesado es inherentemente paralelo: en la etapa de inferencia de resultados permiten optimizar los tiempos de procesado mientras que, el hecho de poder implementar unidades de almacenamiento (memorias) para almacenar los pesos de la red como bloques lógicos dentro de la propia FPGA permite reducir el consumo de potencia, ya que, al evitar conexiones a una RAM (*Random Access Memory*) externa, evita el consumo de la energía necesaria para transmitir las señales al exterior, además de reducir los tiempos de cómputo.

En este documento se presenta el trabajo realizado para implementar sobre una FPGA comercial de bajo coste una red neuronal capaz de combinar las señales procedentes de un array de 16 sensores de gas de una nariz artificial, para dar como salida los niveles de concentración en la atmósfera de dos gases. Esto supone una fase de virtualización de la red por CPU, donde se diseña su arquitectura, seguida de una fase de implementación donde se concretan los detalles del sistema a implementar sobre la FPGA. Finalmente, se ha desarrollado un sistema para poder inferir resultados llamándola desde software.

**Palabras clave:** *Redes Neuronales, FPGA, Machine Learning, Fusión Sensorial, Pytorch, Vivado HLS*

# 1. Introducción

## 1.1. Motivación del trabajo

Sin duda, hoy en día, términos como *inteligencia artificial*, *machine learning (ML)* y *redes neuronales* están en boca de todos. Y no es para menos; la época en la que nos ha tocado vivir nos depara muchas sorpresas en estos campos, los cuales, por su íntima relación con las tecnologías de la información, están presentes en nuestra vida cotidiana. Las redes neuronales son una de las herramientas más utilizadas dentro de lo que se generaliza como inteligencia artificial y, normalmente, están en la base de aplicaciones que usan *machine learning* y que son, muchas de ellas, de uso diario por miles de millones de personas: *Google Translate*, recomendación de videos de plataformas de *streaming*, mejoras y filtros de fotografías, anuncios… Así, una de las motivaciones para llevar a cabo este trabajo ha sido la posibilidad de asomarse a este campo y tratar de aportar algo diferente a las aplicaciones de redes neuronales que todo el mundo conoce, relacionándolo con el ámbito de la sensórica, en concreto con el procesamiento de medidas físico-químicas. En este sentido, diferentes trabajos han sido propuestos con anterioridad para realizar funciones basadas en redes aplicadas a linealización de sensores [2], fusión sensorial [3] [4] o la detección de compuestos orgánicos volátiles (VOCs) [5], entre otros (Figura 1(a)).

Por otra parte, el segundo *boom* de las redes neuronales en los años noventa abordó el desarrollo de este paradigma computacional en paralelo con el estudio de su implementación sobre arquitecturas electrónicas de procesado adecuadas a sus particulares características. Es en esta época cuando se presentan trabajos dónde se implementan redes neuronales en circuitos integrados de lógica programable o FPGAs (*Fiel Programmable Gate Arrays*) [6] [7]. Actualmente, debido al auge que está viviendo el *machine learning*, parece que el uso de redes neuronales en FPGAs vuelve a ser un tema de actualidad, ya que de alguna manera consigue aunar los conocimientos adquiridos en ambos campos en la última década para obtener una solución óptima: por una parte, las herramientas para la descripción, diseño y síntesis de sistemas digitales para la programación de FPGAs; y, por otra, las arquitecturas y algoritmos de optimización del entrenamiento de redes neuronales. Es por esto por lo que creemos que puede ser interesante explorar y combinar estos dos campos, abordando el diseño e implementación sobre una FPGA de una red que determine los niveles de concentración de dos gases en la atmósfera partir de la combinación de las medidas de una nariz artificial de 16 sensores de gas (Figura 1(b)). Este sistema tiene especial interés, ya que podría aplicarse en redes de sensores distribuidos en sistemas ubicuos, evitando realizar un procesado mediante computación en la nube para lo cual sería necesaria una conexión a internet de toda la red distribuida.

El último punto de este trabajo que queremos señalar es su desarrollo empleando software libre. Hoy en día existen numerosas herramientas libres para hacer desarrollo, y una de ellas es el lenguaje de programación Python. Actualmente es el segundo lenguaje de programación más usado en el mundo [8], y dos de las librerías más famosas para *machine learning*, Tensorflow (de Google) y PyTorch (de Facebook) están escritas en Python, por lo que creemos que es interesante tratar de hacer desarrollo con este lenguaje libre.

## 1.2. Objetivos

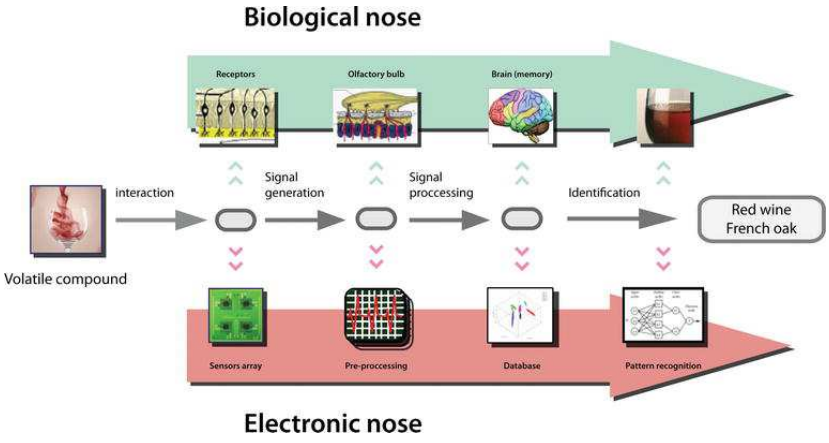
El objetivo principal de este trabajo es implementar una red neuronal en una FPGA, de forma que el diseño de arquitectura electrónica se ajuste lo mejor posible a los requerimientos del modelo neuronal escogido, esto es, optimizando su implementación en términos de minimización de recursos electrónicos, velocidad de procesamiento y respuesta, así como de exactitud en los resultados obtenidos. En concreto, la red neuronal a implementar será capaz de inferir las concentraciones de dos gases a partir de la respuesta procedente de una nariz electrónica consistente en una matriz de 16 sensores de gases. Los objetivos específicos que debe cumplir la red implementada y que, más adelante, servirán para validarla, están relacionados con tres parámetros:

- Tiempo de inferencia. Puesto que las FPGAs pueden implementar estructuras electrónicas de procesamiento en paralelo, son capaces de realizar operaciones recurrentes más rápido que las CPUs convencionales, dando lugar a tiempos de cómputo mucho menores [9] [10]. En nuestro caso, el tiempo que se debe reducir es el tiempo de inferencia, es decir, el tiempo que tarda la red neuronal en devolver una predicción en función de unas entradas. Es de esperar que, para una red neuronal implementada en una FPGA, el tiempo de inferencia se reduzca significativamente.
- Formato y tamaño de los datos. Es bien conocido que, atendiendo a la representación numérica de los datos, existen dos tipos de aritmética: con punto fijo y con punto flotante. Las arquitecturas electrónicas de aritmética en punto fijo son más sencillas que las de punto flotante, pero una representación en punto flotante permite un rango mayor de datos con el mismo número de bits. Este aspecto será tratado muy cuidadosamente, e introduce otro requerimiento: que la representación escogida para los datos no introduzca un error mayor que el asociado a la aproximación funcional proporcionada por la red una vez ajustada.
- Uso de recursos lógicos. Tanto la velocidad de respuesta requerida por el problema como el formato más simple que proporcione la exactitud mínima condicionarán el uso de los recursos lógicos de la FPGA utilizada. Existen métodos de optimización que seleccionan el emplazamiento óptimo del sistema digital en las celdas programables de la FPGA con el fin de reducir la latencia de un diseño, que en nuestro caso corresponde con el tiempo que tarda la red en inferir las salidas una vez el sistema está entrenado para desempeñar su tarea. Por otro lado, el formato de representación de los datos condicionará enormemente el uso de recursos hardware del dispositivo programable. Como ya hemos comentado, la aritmética con representación numérica en punto flotante es mucho más compleja y, por ende, mucho más costosa en cuanto a recursos que la aritmética en punto fijo. Además, también debemos tener en cuenta el número de bits que usemos para describir los datos, ya que un número elevado de bits implicará un mayor uso de área.

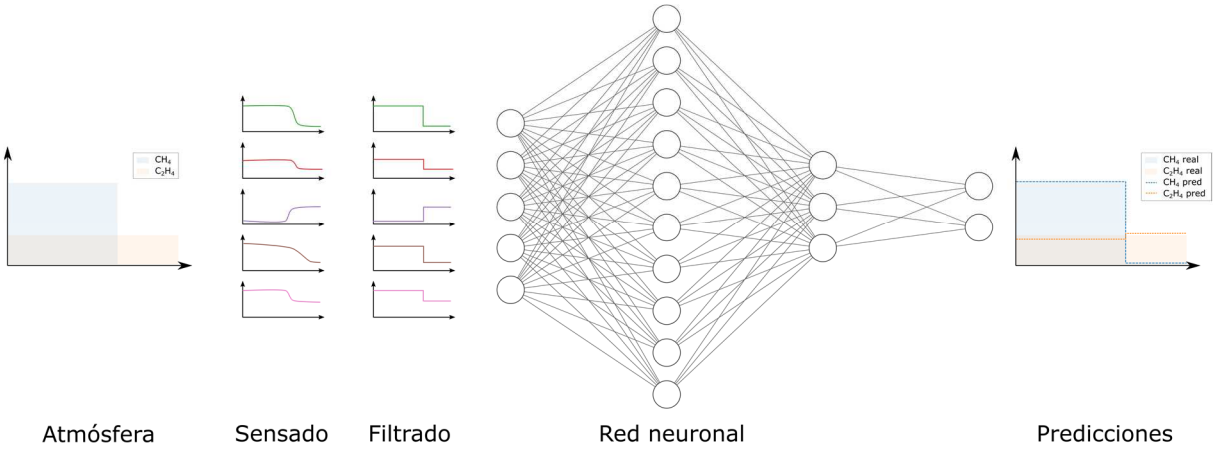
En definitiva, tenemos que encontrar un compromiso entre el tiempo de inferencia, el tipo y tamaño de los datos y el área de FPGA que usemos para implementar la red.

La presente memoria se divide en seis capítulos. El primero presenta la motivación y los objetivos de este trabajo. En el segundo, se presentarán las diferentes tecnologías utilizadas para lograr estos objetivos, tanto a nivel algorítmico, introduciendo las redes neuronales; como a nivel físico, presentando las FPGAs y las distintas formas de representación numérica en aritmética digital. El tercer capítulo tratará sobre la fase de virtualización de la red, realizando un primer estudio de los datos, y discutiremos sobre la arquitectura de la red, presentaremos los resultados de su entrenamiento y analizaremos los efectos de la cuantización de los datos. La fase de implementación y uso se presentará en el capítulo cuarto, donde introduciremos la herramienta utilizada, Vivado HLS, y técnicas de optimización de la red para llegar a su diseño

final, obteniendo resultados sobre el porcentaje de uso de bloques lógicos (ocupación de la FPGA) y tiempos de inferencia. Además, hablaremos de la plataforma PYNQ, que utilizaremos para controlar la FPGA seleccionada, e incluiremos los resultados de la red en términos de detección una vez implementada. En el último capítulo trataremos de recuperar los aspectos más importantes de nuestro trabajo con la presentación de conclusiones, y propondremos posibles líneas futuras.




(a)




(b)

**Figura 1:** (a) Concepto de nariz artificial frente a biológica; (b) Implementación electrónica sobre módulo Pynq con FPGA, y esquema del procesado de datos de la nariz artificial: tras la toma de datos las señales de los sensores pasan por un filtrado para después entrar en la red neuronal, capaz de predecir las concentraciones de dos gases en la atmósfera.

# 2. Implementación de sistemas de ML sobre dispositivos electrónicos reconfigurables

Para determinar los requisitos básicos que debe tener el sistema que proponemos, nos centraremos en dos aspectos esenciales: qué técnica o herramienta de ajuste funcional queremos implementar, y sobre qué plataforma física vamos a hacerlo. En este capítulo nos disponemos a tratar las tecnologías clave que hemos usado para llevar a cabo nuestro trabajo: por un lado, emplearemos las redes neuronales como técnica de ajuste funcional entre los datos de entrada, procedentes de los sensores, y de salida, los niveles de concentración de los gases objetivo. Por otro lado, el soporte físico que emplearemos para su implementación serán las FPGAs. Además, como consecuencia de ambas selecciones, debemos determinar la forma más adecuada en la que representaremos los valores numéricos implicados en el proceso (datos de entrada y salida, parámetros de aprendizaje y coeficientes de ajuste), en términos de error máximo aceptable y complejidad computacional.

## 2.1. Aspectos básicos de las redes neuronales

Se suele entender como red neuronal artificial un tipo de modelo computacional inspirado en las redes neuronales biológicas, en el que cada neurona o procesador elemental está conectada mediante unos enlaces ponderados con otras neuronas. Existen varios tipos de arquitecturas para definir estas conexiones y las operaciones por las que se rigen [1], pero en este trabajo nos vamos a centrar en las redes neuronales *feed fordward fully-connected,* cuya arquitectura nos permitirá una rápida implementación con la capacidad para llevar a cabo adecuadamente la tarea propuesta. Su modelo básico se presenta en la Figura 2. Las neuronas (Figura 2 (a)) están organizadas en capas (capa de entrada–capas ocultas o intermedias–capa de salida), de forma que todas las neuronas de una capa se encuentran conectadas con todas las de la capa anterior y posterior; en el caso de la Figura 2 (b), la red tiene dos entradas, una capa oculta de cuatro neuronas y una salida. En este tipo de modelo neuronal la información es procesada hacia adelante desde la primera capa hasta la última capa, sin conexiones de realimentación de información, y se rige por la siguiente operación entre capas:

$$y^i = f\left(\sum_j w_j^i x^j + b^i\right) \tag{1}$$

donde $y^i$ representa el valor de la neurona $i$ de la capa posterior, $x^j$ el valor de la neurona $j$ de la capa anterior, $w_j^i$ es el peso del enlace entre la neurona $j$ de la capa anterior y la $i$ de la capa posterior y $b^i$ es el peso de un enlace adicional a la neurona $i$, llamado *bias*, cuyo valor de entrada es siempre 1 y constituye un término independiente de las entradas anteriores, dando un grado de libertad adicional al modelo de ajuste. Finalmente, $f(\cdot)$ es la función de

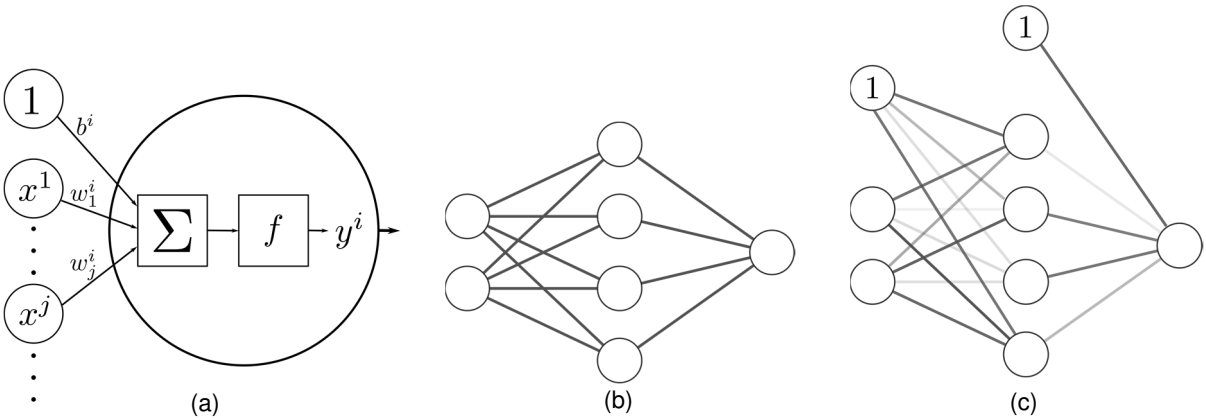



**Figura 2:** Esquemas de redes neuronales. (a) Modelo de una neurona o procesador elemental. (b) Básico, sólo muestra los enlaces. (c) Muestra el peso de cada enlace y las unidades de *bias*.

activación de la neurona. De esta manera, cada neurona $x^j$ proporciona un valor a su salida que condiciona el valor $y^i$ de las neuronas posteriores en función de los valores de los pesos $w_j^i$. En la Figura 2 (c) aparecen los enlaces sombreados según su valor y las unidades *bias*.

Lo que hace especialmente interesantes a las redes neuronales es que se comportan como estimadores no lineales sin necesidad de definir una expresión parametrizada. Por ello, el elemento clave del procesado de una neurona artificial, mostrado en la ecuación (1), es la función de activación que, en definitiva, aporta la no linealidad al ajuste. Existen muchas opciones para esta función, aunque las tres más usadas son las que aparecen en la Figura 3: la tangente hiperbólica, la sigmoide y la ReLU.

Al igual que cualquier estimador, las redes neuronales tratan de ajustar una función atendiendo a cómo de cerca está la función estimada de los puntos a ajustar. Por el modo en que se desarrolla, el proceso de ajuste se conoce como entrenamiento o aprendizaje y los parámetros de ajuste son los pesos $w_j^i$ y *bias* $b^i$. Por su parte, el conjunto de los datos a ajustar se denomina *dataset* y generalmente se divide en dos partes: los datos de entrenamiento (*training set*, 70-80% del conjunto total de datos), que se emplea para ir adaptando la respuesta del sistema a la deseada, y los datos de validación (*validation set,* el restante 30-20% de datos), destinado a verificar la correcta operación del ajuste tras cada iteración del entrenamiento. Para controlar el entrenamiento, se definen diferentes métricas, siendo una de las más comunes el error cuadrático medio (MSE), denominado coste ($\mathcal{J}$), parámetro a minimizar comparando las salidas de la red con los valores del *dataset*. Normalmente, para encontrar el coste mínimo, se usa el descenso de gradiente estocástico (SGD, por sus siglas en inglés), actualizando cada peso proporcionalmente a la derivada parcial del coste del *training set* ($\mathcal{J}_{TS}$) con respecto a ese peso, con un factor de proporción $\eta$

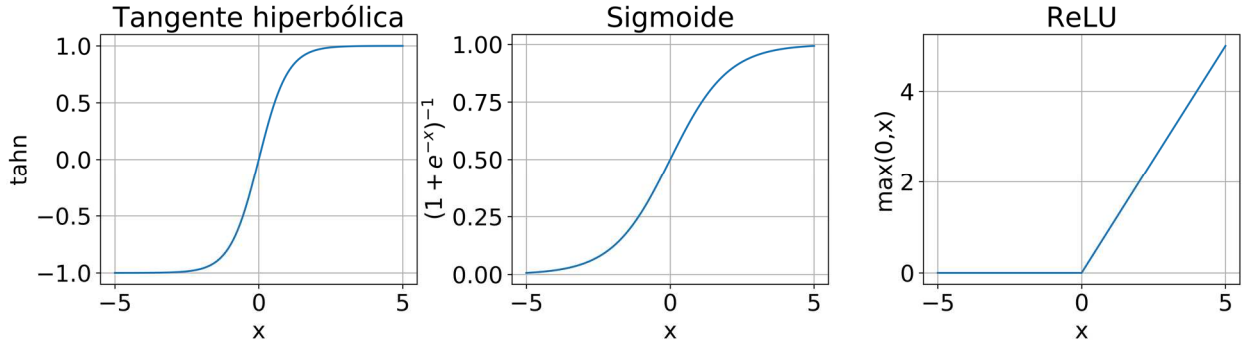



**Figura 3**: Tres de las funciones de activación más usadas: tangente hiperbólica, sigmoide y ReLU.

que corresponde al ritmo de aprendizaje (*learning rate*), es decir, la tasa de aprendizaje en cada iteración del entrenamiento:

$$w_j^i \coloneqq w_j^i - \eta \frac{\partial \mathcal{J}_{TS}(w_j^i)}{\partial w_j^i} \tag{2}$$

Esta es pues la expresión utilizada para actualizar los pesos en base al SGD. El problema principal es calcular eficientemente la derivada del coste con respecto a los pesos, ya que cuando el número de capas es mayor que dos la complejidad aumenta. Esto no fue resuelto hasta 1986 (de ahí el *boom* de las redes neuronales en los años noventa), cuando Rumelhart, Hinton y Williams presentaron el algoritmo de retropropagación del error [11], que se basa en aplicar la regla de la cadena en el cálculo de la derivada, determinando el gradiente de una capa y propagándolo hacia atrás. De esta forma, se consigue evitar cálculos redundantes de términos intermedios de la regla de la cadena. Tras un proceso iterativo de modificación parcial de los pesos en función del error cometido por el sistema en la estimación de sus salidas, y una vez encontrado el mínimo del coste, se dice que la red está entrenada y ya es capaz de inferir correctamente ante nuevos valores de entrada o estímulos.

Como ya se ha comentado antes, una red neuronal es capaz de aproximar cualquier función. El trasfondo matemático es el Teorema de Aproximación Universal, que establece que con una sola capa intermedia una red neuronal es capaz de aproximar cualquier función con una precisión arbitraria, siempre y cuando las funciones de activación sean no polinómicas [12] [13]. Sin embargo, el teorema no indica la forma de conocer el número de procesadores necesarios en esa capa única ni el tiempo de entrenamiento requerido para aproximar una función suficientemente no lineal, lo que hace que sea inviable. Es por esto por lo que, en la práctica, se usan varias capas intermedias, aumentando el número de parámetros de la red y reduciendo de esta manera los tiempos de entrenamiento.

## 2.2. Introducción a las FPGAs

La otra tecnología clave de este trabajo son las matrices de puertas lógicas programables o FPGAs. Básicamente, las FPGAs contienen un conjunto de bloques lógicos configurables (CLB, *Configurable Logic Block*) en una matriz de filas y columnas (Figura 4 (a)), cuya interconexión permite realizar una cierta función lógica, formando sistemas combinacionales y secuenciales, que pueden abarcar desde una simple puerta lógica hasta todo un sistema complejo *on-chip*, como es la red neuronal que pretendemos implementar.

### 2.2.1. Elementos básicos de una FPGA

***Bloques Lógicos Configurables***

Los CLBs están compuestos por distintos elementos (Figura 4 (c)), siendo los más comunes: tablas de consulta (LUT, *Look-Up Table*), *flip-flops (*FF*)* y multiplexores (MUX), mientras que la mejora de prestaciones en los últimos años en las tecnologías de integración ha permitido la incorporación de elementos más complejos a la arquitectura de una CLB, como sumadores y *Digital Signal Processing* (DSP) *slices.*

Una LUT es un sistema digital capaz de realizar cualquier función booleana relacionando las entradas con las salidas mediante una tabla de verdad. Esto las hace especialmente versátiles y son el elemento básico de las FPGAs.

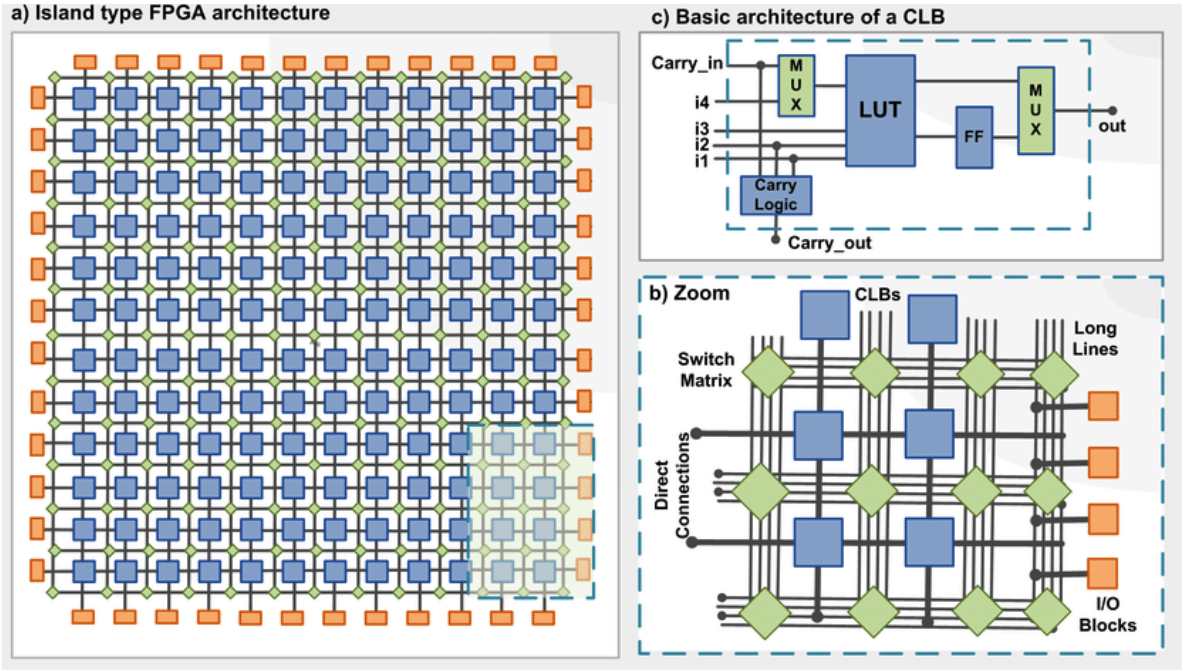
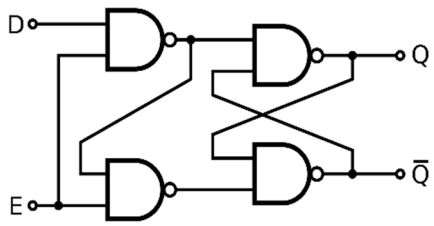



**Figura 4:** (a) Estructura matricial de los bloques lógicos programables de una FPGA, rodeados de líneas de conexión configurables. (b) Detalle de la FPGA, con los bloques de entrada/salida al exterior. (c) Ejemplo de estructura básica de una celda lógica básica. Fuente **[28]**.

Por su parte, los biestables (o *flip-flops*) son elementos secuenciales (cuyo comportamiento depende de varias entradas entre las que se encuentra una señal de reloj) que tienen dos estados estables posibles, siendo por tanto elementos básicos de registro en sistemas electrónicos. Una posible implementación de un biestable de tipo D es el basado en puertas NAND (AND negadas) de la Figura 5.




| E | D | Q | $\overline{Q}$ |
|---|---|---|---|
| 0 | X | Q previo | $\overline{Q}$ previo |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

**Figura 5:** Esquema de la implementación de un *flip-flop* D junto con su tabla de verdad. El símbolo X en la tabla representa que el valor de esa señal es indiferente en esa situación.

La operación aritmética más simple es la suma, por lo que es habitual encontrar en las celdas lógicas de las FPGAs sumadores completos de un bit como el de la Figura 6. Estos sumadores son muy versátiles, ya que son uno de los elementos básicos de casi todas arquitecturas aritméticas.

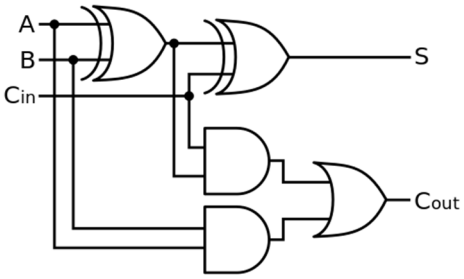



**Figura 6:** Esquema de un sumador completo de un bit.

Uno de los campos donde más éxito están teniendo las FPGAs en los últimos años es el procesado digital de señales, por lo que los fabricantes suelen incluir un elemento extra en los CLBs que se conoce como DSP *slice*. Puesto que en el procesamiento digital de señales (aplicaciones típicas son el procesado de audio y video) es necesario usar recurrentemente multiplicadores y acumuladores, los DSP *slices* cuentan con estas estructuras optimizadas para trabajar a alta velocidad (habitualmente una operación de multiplicación y acumulación ocupa un único ciclo

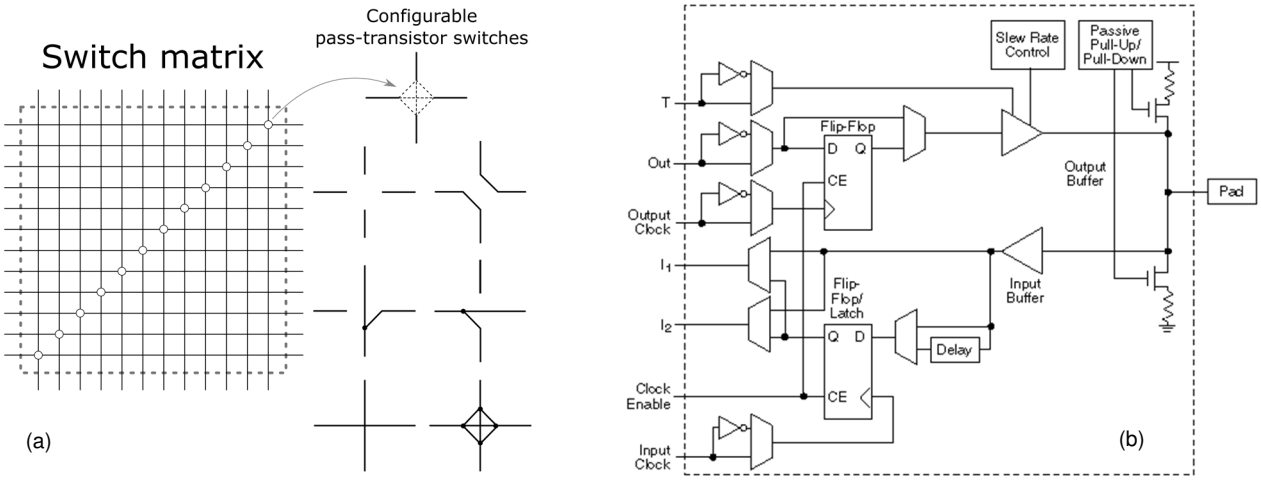



**Figura 7:** (a) Matriz de conexionado o *switch matrix* y posibles conexiones de los switches. (b) Bloque de entrada-salida. Fuente [26].

de reloj) y con un uso de área reducido. Así, gracias a los DSP *slices*, es posible realizar eficientemente operaciones de multiplicación sin necesidad de usar varias celdas lógicas.

***Matriz de conexionado***

Para poder construir el sistema digital deseado a partir de los CLBs de la FPGA es necesario disponer de un sistema que permita interconectarlos adecuadamente. Éste se conoce como matriz de conexionado [14], y es un elemento clave en las FPGAs que consiste, como bien indica su nombre, en una matriz de *switches* (Figura 7 (a)), que habilitan o deshabilitan los caminos de transmisión de señales entre CLBs independientemente de su proximidad física. El diseño geométrico de estas matrices de conexionado, así como las características de los componentes que las forman, condicionan enormemente el tiempo de ejecución de las operaciones en el sistema, determinando su velocidad de operación.

***Bloques de Entrada/Salida***

Obviamente las FPGAs necesitan de un sistema para poder interactuar con el exterior, permitiendo la entrada y salida de señales. Esto se lleva a cabo mediante unos bloques diseñados específicamente para ello, los bloques I/O (*input*/*output*) o de entrada y salida (Figura 7 (b)). Como cualquier interfaz I/O cuentan con dos buffers triestado, uno de entrada y otro de salida, que controlan el modo de operación, y resistencias *pull-up* y *pull-down* configurables, que evitan la aparición de estados indeterminados en ausencia de señales. Para mejorar el sincronismo se colocan *flip-flops* a la entrada y a la salida, donde además se cuenta con un controlador de *slew-rate* para controlar los tiempos de subida y de bajada de las señales.

***Reloj***

Dado que la mayoría de los diseños lógicos son secuenciales, es necesario tener una señal de reloj que marque los tiempos en los que los estados de las señales pueden modificarse. Es habitual que las FPGAs cuenten con varias señales de reloj de diferentes frecuencias, además de PLLs (*Phase-Locked Loops*), bloques electrónicos capaces de generar señales de reloj de frecuencia arbitraria.

***Programación y diseño***

Tan importante como la arquitectura interna de la FPGA es la herramienta disponible para describir adecuadamente la estructura hardware que se quiere implementar. Para describir estos sistemas y las conexiones entre celdas lógicas se utilizan lenguajes de descripción de hardware o HDLs (*Hardware Description Languages*), que permiten trabajar con grandes

estructuras usando descripciones funcionales de alto nivel, evitando así la descripción elemento a elemento de todo el sistema. Los dos HDLs más conocidos son VHDL y Verilog.

Otra opción para programar FPGAs es el empleo de herramientas de síntesis de alto nivel o HLS (*High-Level Syntesis*). Los entornos de HLS interpretan un algoritmo escrito en un lenguaje de alto nivel, como puede ser C++, y lo traducen a un sistema de hardware digital con este comportamiento. En otras palabras, traduce C/C++ a VHDL o Verilog.

La implementación de un diseño en una FPGA tiene tres fases muy marcadas: síntesis, verificación y validación de modelo, y emplazamiento y enrutado (*place and route*). Una vez descrito el sistema mediante un HDL, se debe sintetizar el diseño, lo cual implica identificar los recursos necesarios de la FPGA para ejecutar las tareas descritas en el HDL. En este proceso se pueden utilizar diferentes técnicas de optimización que permiten reducir latencias, uso de área o consumo energético. Finalizada la síntesis, se debe verificar y validar que el diseño cumple con las especificaciones requeridas previamente al paso de *place and route,* donde se determina en qué celdas lógicas se va a implementar el diseño y cómo se van a comunicar estas entre ellas. Existen varias herramientas que asisten al diseñador durante este flujo de trabajo, que explicaremos con detalle en el capítulo cuarto.

## 2.3. Aritmética digital: Punto flotante y punto fijo

Una de las cuestiones básicas en sistemas digitales es la representación de números reales. Como sabemos, dentro de un intervalo acotado, existen infinitos números reales, lo que hace imposible la representación de todos ellos con un número finito de bits. Por ello, es necesario definir unas reglas de representación que describan cuál es la relación entre el código binario y el número real representado. La representación elegida determinará cómo serán las estructuras aritméticas para poder operar convenientemente, lo que repercutirá directamente en su consumo de área en la FPGA.

La más básica es la representación en punto fijo, en la que, como bien refleja su nombre, la posición del punto (el separador entre la parte entera y la parte decimal de un número) es siempre la misma. En esta representación, los $N$ bits totales para codificar un número se separan en dos partes: los de la parte entera, cuyo número de bits denotamos como $I$; y los de la decimal, que ocuparán $B$=$N$-$I$ bits. De esta forma $I$ limitará el intervalo de valores que podemos representar, y $B$ nos dará la resolución máxima en la representación.

Veamos un ejemplo usando $N$=32, $I$=8 y $B$=24 (Figura 8): el número 12,345 está dentro del intervalo, pero no es representable exactamente, por lo que debe codificarse con el

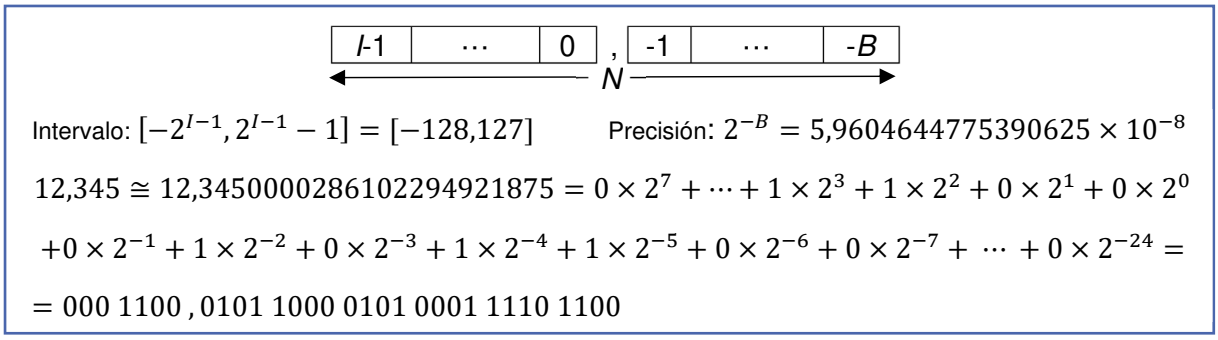



**Figura 8:** Representación de 12,345 en punto fijo usando 8 bits para la parte entera y 24 para la decimal.

representable más cercano, es decir, que sea múltiplo de la precisión que tenemos fijada al elegir *B*. Por tanto, la aritmética asociada a este tipo de representación es semejante a la que se aprende en la escuela primaria, donde las operaciones (sumas, multiplicaciones y divisiones) de números reales se realizan igual que con enteros, posicionando unidades, decenas, centenas… de los números en la misma posición.

Por otro lado, está la representación numérica en punto flotante. Probablemente fue propuesta por primera vez por Leonardo Torres y Quevedo en 1913 [15], cuando trabajaba en un aritmómetro (precursor electromecánico de una ALU, unidad aritmético-lógica, Figura 9). Torres y Quevedo se planteaba el problema de representar números muy grandes, ya que implicaba un uso elevado de recursos[1] cuando realmente sólo eran útiles un número reducido de ellos, los que se correspondían con las cifras significativas. Es por ello por lo que él propone el uso de una representación que hoy en día se conoce como la de punto flotante, en la que las cifras significativas (significante o mantisa) se separan del orden de magnitud (exponente). El estándar actualmente es el IEEE 754; define que un flotante simple (*float* de C++, por ejemplo) utiliza un total de 32 bits, de los cuales uno es para el signo de la mantisa, 23 para el valor de esta y 8 para el exponente.

Veamos cómo se representaría el ejemplo anterior en esta notación (Figura 10): en punto flotante 12,345 es igualmente no representable, y la diferencia con el valor representado es mayor, algo lógico ya que, con esta representación, si empleamos el mismo número de bits, se consigue un mayor rango, a costa de reducir la resolución de la representación. La conversión es bastante más complicada, por lo que no entraremos en más detalles sobre ella. Lo que queremos destacar son las implicaciones en las estructuras aritméticas necesarias para los cálculos ya que, al tener los datos representados de esta forma, es necesario operar por un lado con las mantisas y por otro con los exponentes, alineándolos adecuadamente, lo

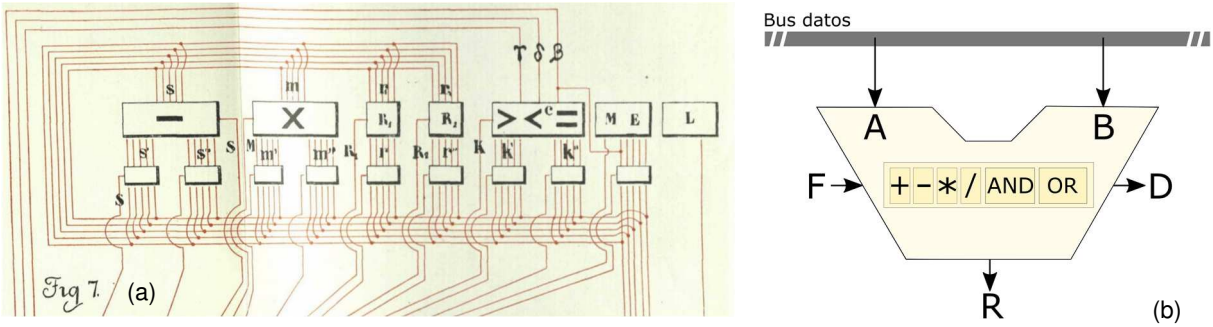



**Figura 9:** (a) Recorte de una de las figuras de la publicación de Torres y Quevedo [15]. Es apreciable la semejanza de este esquema de un aritmómetro electromecánico con el esquema que tendría una ALU hoy en día (b).

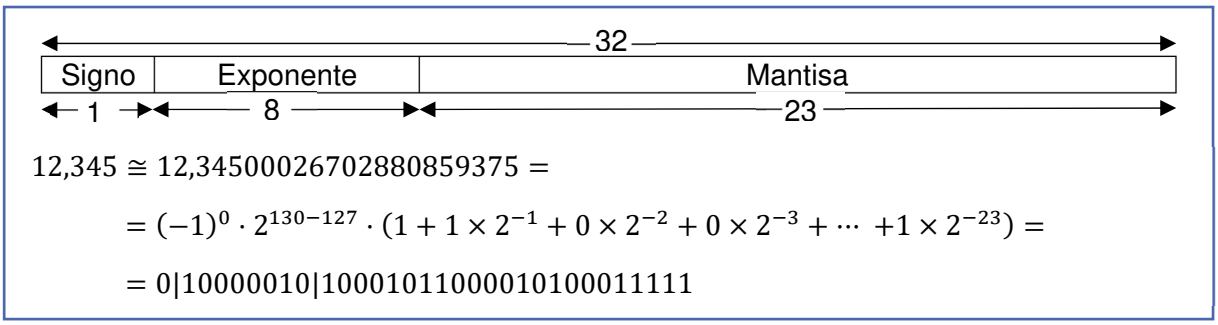



**Figura 10:** Representación de 12,345 en punto flotante de 32 bits usando el estándar IEEE 754.

[1] Cuando hablamos de recursos, en la actualidad debe entenderse como número de bits. En la época de Torres y Quevedo la representación se realizaba mediante combinaciones de conmutadores, lo que dota al concepto de recurso de un sentido mucho más costoso.

que complica considerablemente la electrónica, que necesitará de más bloques lógicos que los que pudiésemos necesitar usando punto fijo.

Así pues, volviendo al objetivo principal de este trabajo, la implementación de una red neuronal en una FPGA, existen varios estudios en los que se evalúa el efecto del uso de representación en punto fijo con un $N$ reducido en redes neuronales [16] [17]. Es obvio que la pérdida de resolución debido a la reducción de bits afecta a la red, pero estos efectos se pueden minimizar reentrenando la red, llegando a conseguir resultados sorprendentes. De esta forma, se consigue reducir significativamente el uso de área, debido a que los datos ocupan menos bits (registros menores) y además las estructuras aritméticas son más sencillas (paso de punto flotante a punto fijo).

# 3. Fase de virtualización

En este capítulo vamos, en primer lugar, a presentar el problema que ha de resolver nuestra red neuronal y mostraremos el conjunto de datos empleado. A continuación, realizaremos una breve descripción de la arquitectura de la red y abordaremos su entrenamiento. Por último, analizaremos los efectos del uso de la representación en punto fijo en este entrenamiento.

## 3.1. Descripción del problema

Como ya se ha comentado en la introducción, las redes neuronales son estimadores funcionales no lineales que no necesitan la expresión analítica de la función a ajustar. Esto las hace especialmente interesantes para usarlas en procesos de combinación de varias salidas de sensores: la fusión sensorial. En estas aplicaciones, se entrena un modelo neuronal para que, a partir de las respuestas eléctricas de un conjunto determinado de sensores, la red proporcione como respuesta una información combinada de estas. Aplicaciones de fusión sensorial basadas en redes son, por ejemplo, la determinación de contaminantes en atmósfera o la detección de compuestos orgánicos volátiles (VOC) derivados de procesos biológicos (maduración de manzanas, clasificación de vinos, etc.) [5], [18] [19].

En nuestro caso, el objetivo es entrenar una red neuronal para estimar la concentración de dos gases, a partir de las salidas de 16 sensores químicos diferentes. En la práctica, esto es especialmente interesante ya que, si los volátiles son de características parecidas, la selectividad de los sensores no permite diferenciarlos con exactitud y, por tanto, la estimación de la concentración de cada uno de ellos requiere un post-procesado complicado, siendo en muchos casos imposible su discriminación. Para ello, se dispone una configuración matricial de varios transductores sensibles a las magnitudes que se pretende identificar, cada uno con diferente selectividad para cada uno de los gases, de forma que mediante la interpretación combinada de sus respuestas (fusión sensorial) se puede obtener la información buscada.

Los datos utilizados, obtenidos en [20], y disponibles en el repositorio del *Center of Machine Learning and Inteligent Systems* de la *University of California, Irvine* (UCI) [21], contienen las series temporales de dos experimentos independientes con mezclas de gases diferentes. Así, tenemos dos *datasets*: uno corresponde a una mezcla de etileno ($C_2H_4$) y metano ($CH_4$), y en el otro la mezcla es de etileno y monóxido de carbono (CO).

Son dos archivos de texto, uno para cada mezcla, que especifican para cada instante de tiempo las concentraciones de los dos volátiles y la repuesta digitalizada de los 16 sensores. Las medidas se realizan a intervalos de entre 80 y 120 segundos, fijados aleatoriamente. Los sensores empleados corresponden a cuatro modelos distintos de *Figaro Inc.*: TGS-2600, TGS-2602, TGS-2610, TGS-2620 [22], de forma que la matriz de sensores consta de cuatro transductores de cada modelo. La Figura 11 reproduce las respuestas de estos sensores a diferentes gases en función de la concentración; podemos apreciar que para volátiles parecidos la respuesta es similar.

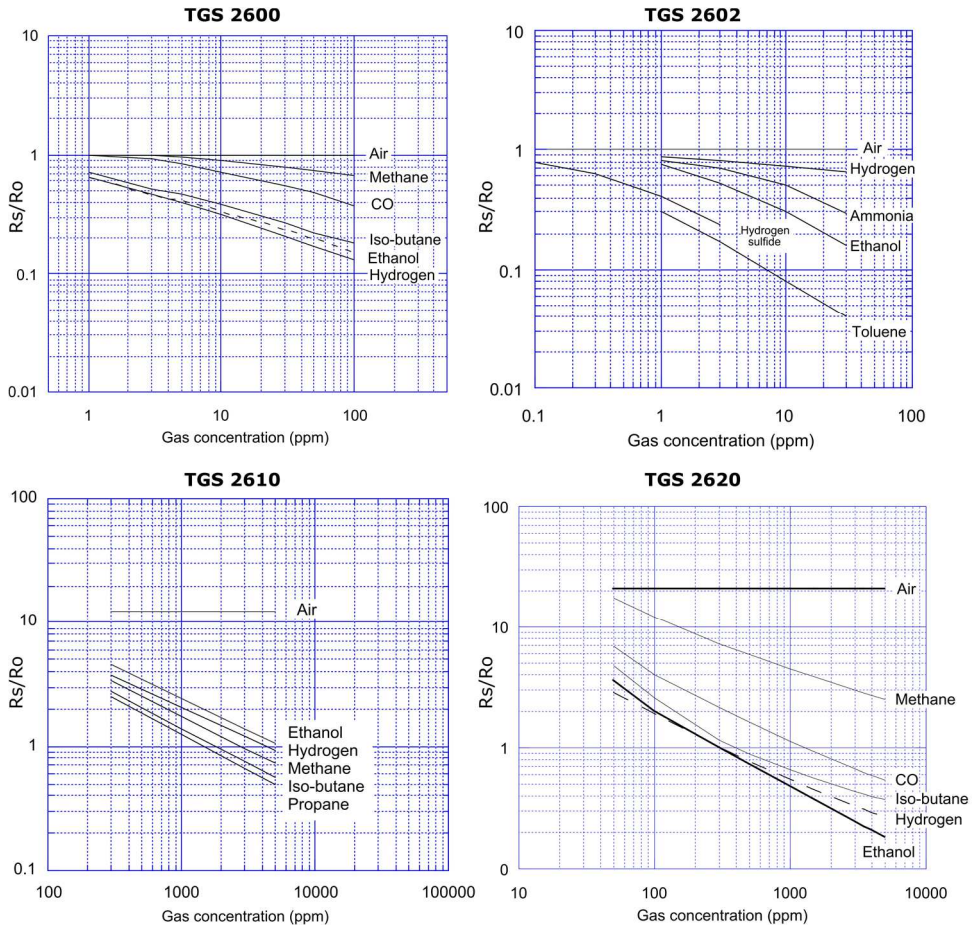



**Figura 11:** Respuesta de los sensores TGS-2600, TGS-2602, TGS-2610 y TGS-2620 (respectivamente) en función de la concentración de diferentes gases. Fuente [22].

De las dos mezclas disponibles, para realizar todo el proceso de análisis de los datos, entrenamiento de la red e implementación en la FPGA usaremos en este trabajo la correspondiente a etileno y metano, dejando las medidas de la otra mezcla para realizar la validación del flujo de trabajo.

## 3.2. Acondicionado de los datos

El primer paso es analizar los datos de nuestro *dataset* para determinar su resolución temporal y, si es posible, establecer una relación directa entre las salidas de los sensores y los correspondientes valores de concentración. Las series temporales que emplearemos están compuestas por un total de 12 horas de medida. Sin embargo, a modo de ejemplo, vamos a representar los datos de una ventana temporal reducida pero que permita apreciar con claridad la necesidad de cada uno de los pasos siguientes que vamos a dar de cara a su acondicionamiento. Esta ventana temporal de ejemplo es la de la Figura 12, donde se muestra la evolución de la concentración de metano y etileno y las correspondientes salidas de los 16 sensores en un periodo de 2000 segundos. En la figura se puede apreciar que existe cierta relación entre las salidas de los sensores y las concentraciones de los gases, pero no tenemos una relación biyectiva, es decir, para cada combinación de concentraciones no existe una única salida de cada sensor. Esto se debe tanto a que los sensores de gas tienen una respuesta dinámica (y, por tanto, un cierto tiempo de respuesta), como a su falta de especificidad para cada gas (es decir, la respuesta del sensor viene condicionada por su

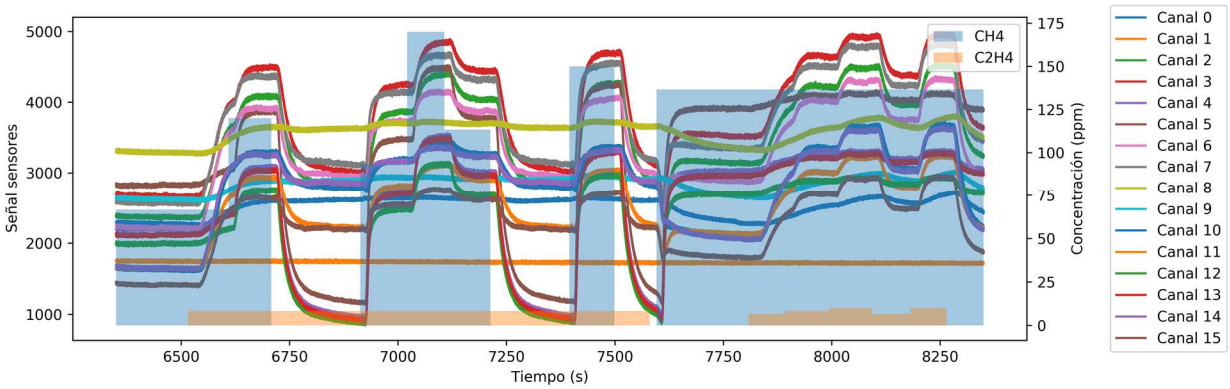



**Figura 12:** Ventana temporal de los datos directos del *dataset*, en donde se muestran las salidas de los sensores digitalizadas y las concentraciones de $CH_4$ y $C_2H_4$ para cada instante de tiempo.

selectividad, que hace que detecte la concentración de los dos gases de forma indistinguible). Por otro lado, los rangos de salida de los sensores son bastante dispares, algo que podíamos esperar a partir de las respuestas mostradas en la Figura 11.

Por todo ello es necesario realizar una normalización y pre-procesado de los datos, que permitan obtener una relación entradas-salidas más unívoca, que nos beneficiará al abordar al entrenamiento de la red.

### 3.2.1. Normalización

Este proceso busca transformar todos los datos de nuestro *dataset* para que estén dentro del intervalo [0,1]. Aunque teóricamente no es necesario según el Teorema de Aproximación Universal, en la práctica es muy recomendable ya que reduce los tiempos de entrenamiento significativamente. Esto se debe a que, en el caso de tener una entrada de la red en un rango muy diferente a las demás, el sistema puede darle a priori a dicha entrada una importancia relativa mayor, degradando el rendimiento de la red; se trata de que el sistema sea capaz de ajustar mediante el entrenamiento la importancia de cada entrada, para lo cual es conveniente que todas partan de una situación equitativa.

Además, en nuestro caso es especialmente interesante normalizar los datos, ya que si tuviéramos entradas con diferentes órdenes de magnitud necesitaríamos un número de bits $N$ muy elevado para poder hacer la representación en punto fijo, algo que repercutiría perjudicialmente en las necesidades de recursos lógicos.

La normalización de los valores se hará de acuerdo a la siguiente fórmula:

$$\overline{x} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3}$$

donde $x$ es el valor de la magnitud medida, $x_{max}$ su valor máximo dentro del rango, $x_{min}$ el mínimo y $\overline{x}$ la magnitud normalizada. Realizamos este proceso con las dos concentraciones y las 16 salidas digitalizadas de los sensores. Como podemos ver en la Figura 13, así los datos comparten el mismo orden de magnitud y que están en el rango [0,1].

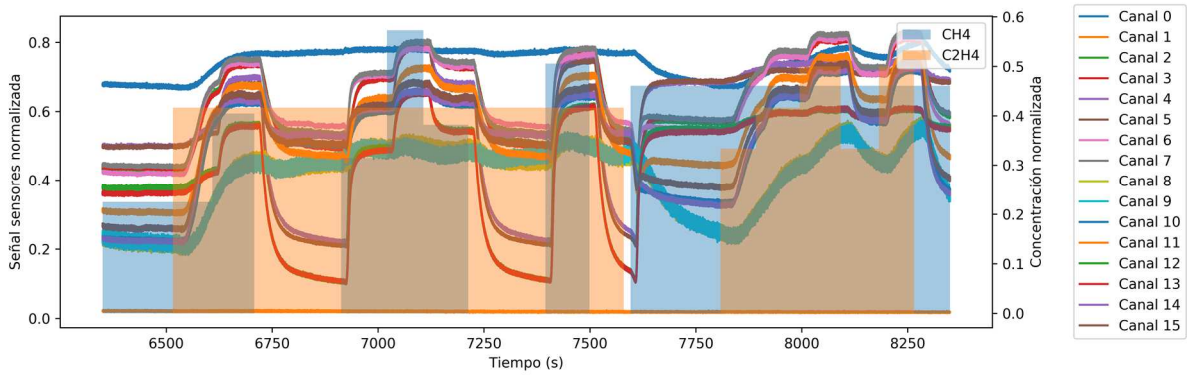



**Figura 13:** Ventana temporal de la serie de datos en la Figura 12 una vez normalizados.

## 3.2.2. Tratamiento de las señales de los sensores

Como se puede observar en la Figura 13, las respuestas de los sensores no son instantáneas frente a los cambios de concentración. La limitada velocidad de respuesta de los sensores se traduce en un tiempo en el que la salida eléctrica aumenta (o disminuye) de forma exponencial tras el incremento (o decremento) de la concentración de los gases. Para conseguir una relación adecuada entre las concentraciones de los gases y las salidas de los sensores, procesaremos éstas de modo que proporcionen un valor único para cada sensor en cada periodo de tiempo en que las concentraciones se mantienen constantes. Para ello encontramos dos posibilidades, que se corresponden con dos formas de medir las concentraciones de los gases.

La primera opción es hacer un promediado en la parte derecha de las ventanas temporales en las que las concentraciones son constantes, cuando los sensores muestran tendencia asintótica a la estabilidad en su respuesta (Figura 14). De esta manera, conseguiríamos emular un comportamiento estático de los sensores, consiguiendo que la red entrenada a partir de los datos con un filtrado de este tipo devolviese las concentraciones de los volátiles cuando el tiempo de medida es suficientemente largo (unos 80 s). La segunda opción es emular el comportamiento dinámico de los sensores, usando la derivada de sus salidas justo después del cambio de concentración, suponiendo que cambios mayores en el nivel de concentración implican mayores pendientes en su respuesta (Figura 14).

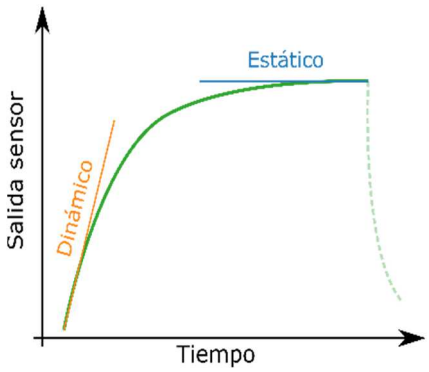



**Figura 14:** Dos opciones de filtrado, dos comportamientos del sensor.

Puesto que lo habitual es que los cambios en concentración de los gases sean más lentos y no tan escalonados como los que nos encontramos en el *dataset*, y que el comportamiento estático permite una medida continua de las concentraciones, nos hemos decantado por usar un tratamiento que reproduzca este comportamiento. Con este fin, vamos a realizar un promediado del 10% de los últimos valores medidos en cada ventana temporal, lo que nos permite obtener una estimación del valor asintótico de la lectura del sensor. Una vez realizado este filtrado, nuestra ventana temporal de ejemplo se muestra en la Figura 15.

El código desarrollado para llevar a cabo toda esta sección de procesado de los datos se puede encontrar en el Anexo I.

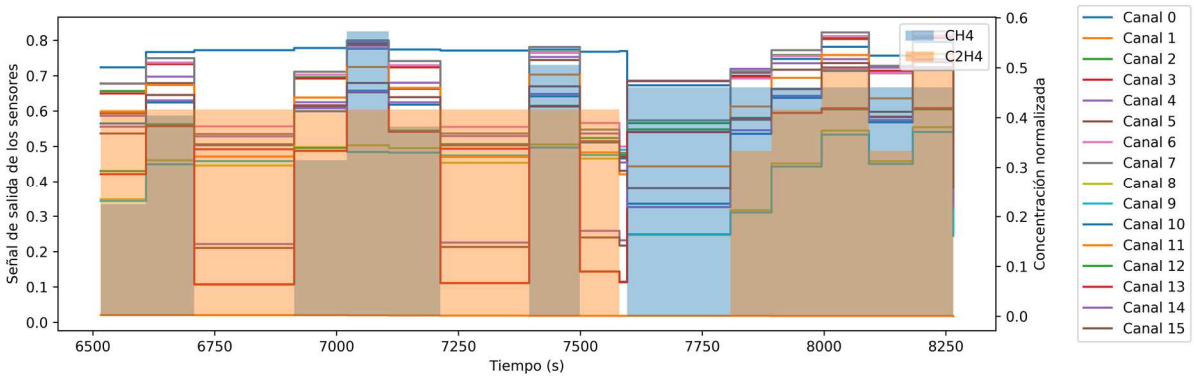



**Figura 15:** Ventana temporal con los datos filtrados.

### 3.2.3. Selección del *training set* y del *validation set*

Una vez que ya tenemos los datos, debemos elegir cuáles de ellos van a usarse para el entrenamiento y cuales para la validación, es decir, debemos separar el *training set* del *validation set*. Para ello vamos a considerar una partición típica en aplicaciones de redes neuronales, con una separación del 70% de los datos originales para el *training set* y el 30% para el *validation set*.

Algo que debemos tener en cuenta, además, es el orden en el que se presentan los datos en el proceso de ajuste de la red, ya que no es lo mismo realizar el entrenamiento presentando los datos siempre en el mismo orden que desordenados aleatoriamente en cada iteración del proceso de aprendizaje. En principio, según la descripción del *dataset*, los datos son aleatorios porque las concentraciones se fijan de esta manera. Aun así, hemos decidido desordenar aleatoriamente los datos del *training set* en cada época del entrenamiento. Por su parte, los datos del *validiation set* no es necesario desordenarlos, ya que su objeto es cuantificar el estado del entrenamiento para establecer cuándo pararlo en función de su evolución (error de ajuste en esos datos). Esta parte del código se muestra en el Anexo II.

## 3.3. Diseño de la arquitectura de la red neuronal

Debemos definir una arquitectura de red neuronal que permita llevar a cabo la función para la que la queremos. Esto implica elegir un tipo de red y sus características: número de capas, número de neuronas por capa y función de activación. Como ya comentamos en el capítulo introductorio, la red que vamos a usar es *feed fordward fully-connected*, el tipo más básico de red, pero suficiente para nuestro propósito. En cuanto a las características, algo que tenemos fijado por el *dataset* es el número de neuronas de la primera y última capa, ya que se van a corresponder con el número de entradas y de salidas, respectivamente; así, tendremos 16 neuronas de entrada, una por cada salida de los sensores de gas, y dos de salida, que nos darán los valores estimados para la concentración de metano y etileno. Con respecto al número de capas ocultas (capas intermedias) y el número de neuronas de cada una de ellas, realmente no existe ninguna regla teórica que dicte cuáles son los valores óptimos y generalmente se usan reglas empíricas, como elegir un número arbitrario y comprobar cuál es su funcionamiento, que generalmente se mide con el tiempo de entrenamiento necesario hasta alcanzar el objetivo de error. En nuestro caso, hemos probado varias configuraciones hasta llegar a la 16-32-12-2, que consigue un buen compromiso entre el número de

parámetros y el tiempo de entrenamiento, el cual ha sido de unos 2 minutos sobre una CPU Intel i5-3470 a 3,2 GHz. En esta configuración contamos con dos capas ocultas de 32 y 12 neuronas respectivamente, de ahí el nombre 16-32-12-2, lo que hace que en total tengamos 4 capas con esos números de neuronas. La idea de esta configuración es expandir las entradas de 16 a 32, buscando posibles combinaciones de éstas para más tarde comprimirlas con los saltos 32-12 y 12-2, lo que permite una transición más suave que realizarla de golpe en un salto 32-2. Con respecto a la función de activación vamos a usar la logística (Figura 3, centro), de modo que conseguimos que los valores queden acotados dentro del rango (0,1) que hemos establecido al normalizar; con esto conseguimos dos puntos importantes: mantener una mayor no linealidad que la ReLU, facilitando un entrenamiento más rápido; y evitar valores de salidas fuera de este rango que impliquen el uso de un mayor número de bits en su representación en punto fijo. Por otro lado, puede ser que la elección de la logística como función de activación complique su implementación en la FPGA, aunque esto lo discutiremos más adelante.

El código de PyTorch usado para esta sección aparece en el Anexo III.

## 3.4. Selección de los parámetros de entrenamiento

Una vez que tenemos preparados los datos de entrenamiento y los datos de validación, y definida e inicializada la arquitectura de red, podemos comenzar con su entrenamiento, la parte más interesante de cualquier trabajo con redes neuronales. Para ello, es necesario elegir correctamente el algoritmo de entrenamiento y las métricas que nos van a permitir cuantificar su progreso. En el capítulo 2 de presentación de las redes neuronales mencionamos que el algoritmo típico para la actualización de los pesos es el descenso de gradiente estocástico (SGD), usando el error cuadrático medio (MSE) como métrica para computar el coste. El MSE lo vamos a mantener, ya que es una métrica válida para cualquier problema de regresión y se suele usar por defecto, pero en lugar de usar el SGD optamos por Adam, una versión optimizada presentada en 2015 [23]. Adam es una variación del SGD que optimiza el entrenamiento utilizando diferentes reglas de actualización del ritmo de aprendizaje mediante el gradiente y el laplaciano del coste, lo que lo hace mucho más eficiente reduciendo el tiempo de entrenamiento. Aun así, todavía es necesario definir el ritmo de aprendizaje que usará Adam como base, fijado en nuestro caso a $10^{-4}$, un valor típico para este parámetro.

Con el optimizador y la métrica definidos, ya podemos entrenar la red. Para ello definiremos el número máximo de pasos de iteración del proceso, que establecerá el momento de finalización del entrenamiento. En cada iteración obtendremos las predicciones de la red para cada uno de los elementos del *training set* y las compararemos con los valores reales de las concentraciones (los valores objetivo) de cada uno de ellos. Con esto obtendremos el MSE de la red en esa iteración, el cual usaremos para retropropagar y actualizar los pesos usando Adam. De esta forma, tras alcanzar el número máximo de iteraciones, que hemos fijado en $10^4$, conseguimos reducir el coste del *training set* y del *validation set* hasta que este último comienza a reducir su ritmo de disminución, lo que indica que el sistema comienza a memorizar los datos y perder capacidad de generalización, momento adecuado para concluir el entrenamiento. La Figura 16 muestra la evolución del coste en los dos conjuntos de datos así como las predicciones de la red durante la ventana temporal de ejemplo tras su entrenamiento. Como se puede ver, las predicciones son bastante cercanas a las concentraciones reales. Más adelante trataremos de cuantificar estos errores.

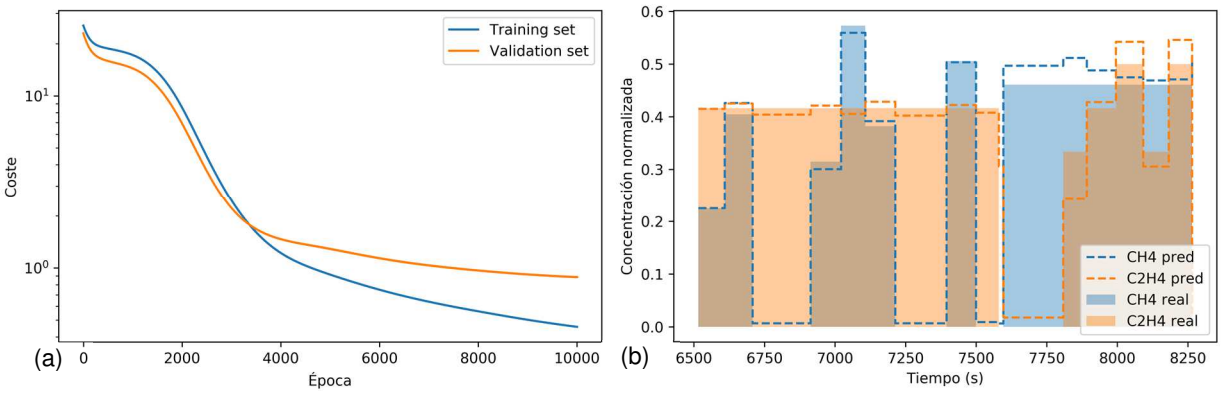



**Figura 16:** (a) Evolución del coste del *training set* y del *validation set* durante el entrenamiento y (b) predicciones de la red para la ventana temporal de ejemplo comparado con las concentraciones reales de los gases.

En el Anexo IV se encuentra el código de esta sección.

# 3.5. Cuantización del entrenamiento

Para poder determinar una solución que pueda implementarse con la menor complejidad posible sobre un dispositivo FPGA, en primer lugar determinamos la menor resolución de los valores numéricos implicados en la operación de la red neuronal que permita obtener unos resultados aceptables en términos de error permisible, o función de coste. Para ello, introducimos el efecto de la cuantización de los pesos en el entrenamiento, ya que esto nos permitirá estimar los errores de la red neuronal implementada sobre la FPGA en función de la representación en punto fijo que elijamos para los pesos. Así, hemos implementado la red neuronal usando los pesos obtenidos en punto flotante tras esta primera fase del entrenamiento, y hemos obtenido las salidas que la red tendría como consecuencia de diferentes representaciones de punto fijo de los datos. Los resultados obtenidos se muestran en la Figura 17, donde los tipos de dato en punto fijo se identifican usando la notación *N-I*

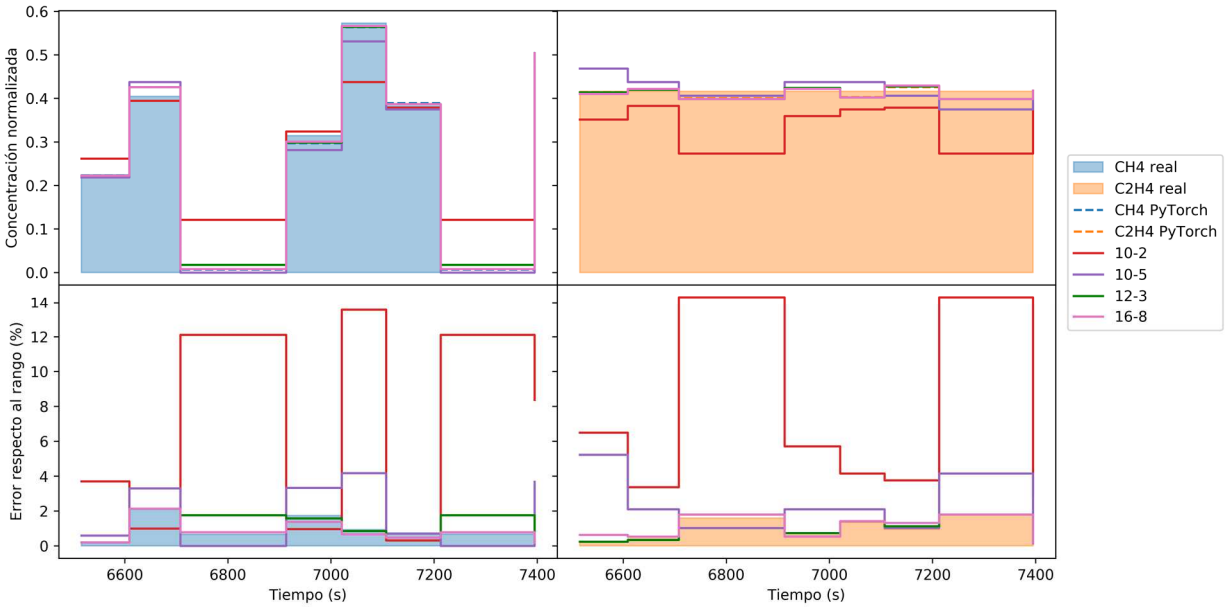



**Figura 17:** Predicciones de la concentración de cada gas durante la ventana temporal de ejemplo para diferentes tipos de punto fijo, los cuales están nombrados según la notación *N-I*, junto con el error obtenido para cada caso.

mostrada en la Sección 2.3. Como se ve en la figura, el formato de dato en punto fijo que mejor se adapta al compromiso entre el error y el número de bits es el 12-3, ya que su respuesta es bastante cercana a la que se obtiene directamente de PyTorch, que usa una representación de punto flotante de 32 bits; esto nos lleva a seleccionarlo como tipo de dato con el que cuantizar los pesos en la segunda fase del entrenamiento.

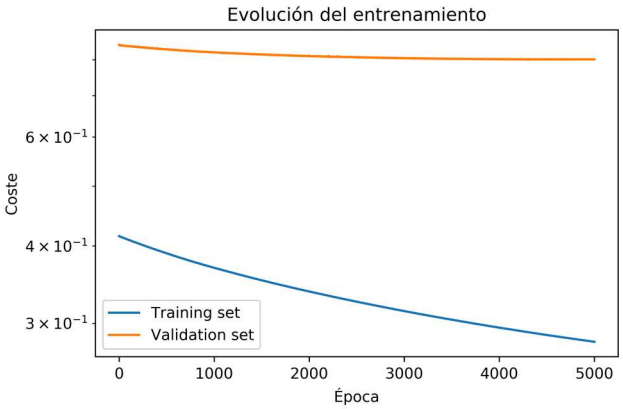



**Figura 18:** Evolución de coste del *training set* y del *validation set* durante la segunda fase del entrenamiento en la que se cuantizan los pesos usando punto fijo 12-3.

Para simular la operación de la red con representación numérica en punto fijo, hemos modificado el código de entrenamiento que usamos, para que discretice los pesos en el formato 12-3 escogido cada 100 iteraciones, tal y como se puede observar en el Anexo V. De esta forma, partiendo del modelo entrenado en la primera fase, en punto flotante con máxima resolución numérica, podemos ver la evolución de los valores de las métricas tras la cuantización, hasta que el coste del ajuste para el conjunto de datos de validación se estabiliza por completo (Figura 18). En este punto damos por finalizado el entrenamiento, ya que seguir el proceso podría dar lugar a sobreajustar los datos, perdiendo la capacidad de generalización de estos sistemas.

### 3.5.1. Resultados del entrenamiento con cuantización de parámetros

Antes de implementar en el siguiente Capítulo el diseño de la red sobre una FPGA, vamos a analizar más detalladamente los resultados del entrenamiento obtenidos en esta Sección y justificar, dentro de lo posible, los errores de la predicción frente a los datos reales de las concentraciones de los volátiles bajo estudio. Para ello, vamos a definir el error relativo $\varepsilon_i$ en la estimación de la concentración del volátil $i$, respecto al rango de las concentraciones, de la siguiente manera:

$$\varepsilon_i = \frac{y_i' - y_i}{y_{i_{\max}} - y_{i_{\min}}} \cdot 100 \tag{4}$$

donde $y_i'$ es su concentración estimada, $y_i$ su valor real, $y_{i_{\max}}$ el máximo del rango e $y_{i_{\min}}$ el mínimo. Por supuesto, nuestras concentraciones han sido normalizadas, por lo que si introducimos la expresión (3) tenemos que:

$$\varepsilon_i = \frac{(\overline{y_i'} - \overline{y_i})(y_{i_{\max}} - y_{i_{\min}}) + y_{i_{\min}}}{y_{i_{\max}} - y_{i_{\min}}} \cdot 100 \tag{5}$$

donde $\overline{y_i'}$ e $\overline{y_i}$ representan las concentraciones normalizadas estimada y real, respectivamente. Usando esta métrica, obtenemos el resultado de la Figura 19 en el conjunto de datos perteneciente a la ventana temporal de ejemplo, donde se observa que los errores, en promedio, son inferiores al 2%.

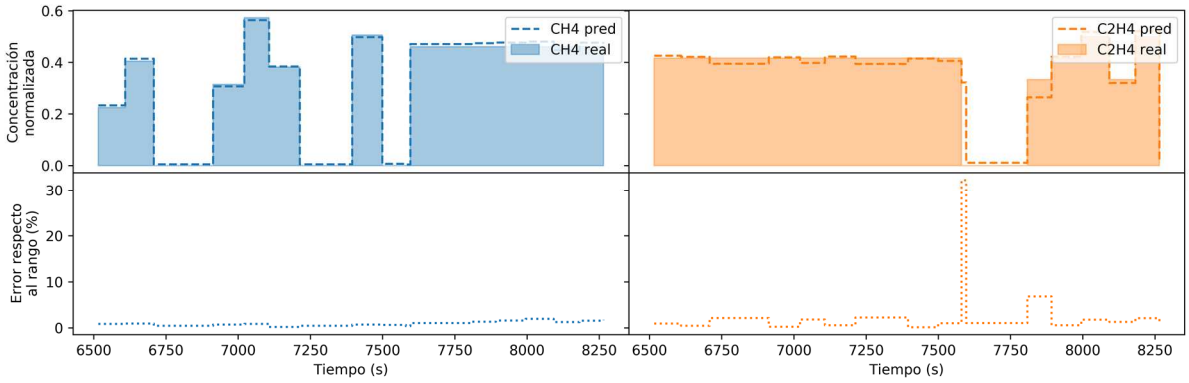



**Figura 19:** Predicciones de la red para los datos dentro de la ventana temporal de ejemplo comparadas con las concentraciones reales de los dos gases. También se presentan los porcentajes de error respecto al rango.

Para realizar un análisis general de los errores, consideramos la Figura 20 (a), donde además de los errores para cada concentración se representan dos líneas discontinuas. La línea horizontal se corresponde con un cambio de tipo de error; en general, los errores por encima del 10% se pueden asociar a cambios bruscos en la concentración del gas (Figura 20 (b)), por debajo de los 80 s que limitaba la descripción del *dataset*. Esto hace que los sensores no tengan tiempo suficiente para estabilizarse y, por tanto, la técnica de filtrado que hemos aplicado pierda estos cambios y la red no sea capaz de percibirlos.

El resto de los errores, los que no se deben a singularidades del conjunto de datos, por debajo del 10%, algo bastante razonable teniendo en cuenta que nuestro histórico de datos es relativamente pequeño. A pesar de ello, observamos que el rango de los errores en la estimación de los gases crece conforme su concentración aumenta, tal y como se aprecia en la otra línea discontinua presentada en la Figura 20 (a). Esto puede deberse a dos factores: por un lado, el limitado número de medidas de alta concentración existente para los dos gases, que afectan la eficiencia del entrenamiento de la red para esos patrones, aumentando el error en la estimación; y la caída de la sensibilidad de los sensores para concentraciones altas, tal y como se ve en la Figura 11.

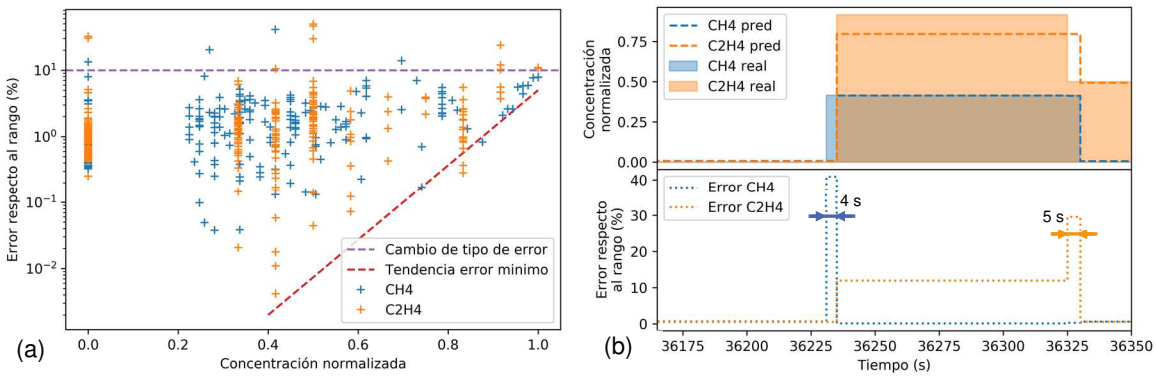



**Figura 20:** (a) Error para cada concentración en donde se marcan dos líneas: la del cambio de tipo de error y la tendencia del error minimo. (b) error debido a defectos en el dataset, por encima del 10%.

# 4. Fase de implementación y uso

En el capítulo anterior hemos diseñado una red neuronal capaz de predecir las concentraciones de los gases en función de las salidas de los sensores de una nariz artificial y, además, hemos ajustado los pesos del modelo neuronal teniendo en cuenta que estos deben estar cuantizados a un formato de representación de punto fijo. En este capítulo vamos a trasladar la red a la FPGA utilizando un lenguaje HDL, concretamente HLS. Más adelante, usaremos el diseño de la red descrito mediante HLS para generar un archivo *bitstream* (encargado de programar los switches de la matriz de conexionado que se activan para fijar la arquitectura final del circuito digital) el cual nos servirá para configurar la FPGA. Finalmente, presentaremos el controlador y el uso de la red implementada en la FPGA durante la fase de explotación.

## 4.1. Diseño en HLS

Como ya comentamos en la Sección 2.2, entre todas las posibilidades que existen para poder programar una FPGA, HLS es la opción que permite hacerlo describiendo el sistema desde un punto de vista algorítmico, usando código en C/C++. Si bien también sería posible realizar una descripción más cercana al hardware de la red usando VHDL o Verilog, creemos que, dada la complejidad del sistema a implementar, HLS va a permitir una descripción de alto nivel que más tarde podremos ir depurando, incluyendo las directrices de hardware necesarias.

Así pues, en primer lugar vamos a presentar la herramienta Vivado HLS, incluida en el paquete WebPack de Xilinx y que permite simular, sintetizar y verificar diseños escritos en HLS para más tarde exportarlos. En la fase de simulación la herramienta se encarga de compilar, ejecutar y depurar el código escrito en C/C++. Una vez completado este paso, en el proceso de síntesis se identifican las operaciones aritmético-lógicas necesarias para llevar a cabo la solución descrita en el código desarrollado. Dichas operaciones son programadas sobre la estructura de la FPGA seleccionada para que se ejecuten en un determinado ciclo de reloj y para que se mantenga la coordinación temporal de las operaciones, y se les asignan los recursos concretos necesarios dentro de los CLBs del dispositivo, así como de conectividad entre bloques. Finalmente, en la fase de verificación se realiza una simulación basada en el hardware asignado, validando que las salidas de cada una de ellas es la que se espera en cada instante temporal.

Como vemos, Vivado HLS incluye todos los pasos a seguir en el flujo de trabajo para el diseño en FPGAs, exceptuando el de generar el *bitstream*, el cual generaremos más adelante con el programa principal de Vivado.

### 4.1.1. Simulación

Obviamente, el paso previo una vez que conocemos la herramienta es transformar nuestro modelo de red neuronal en PyTorch en uno descrito en C/C++. La ventaja de haber usado redes neuronales *feed-forward* es que su transformación no resulta compleja: contamos con

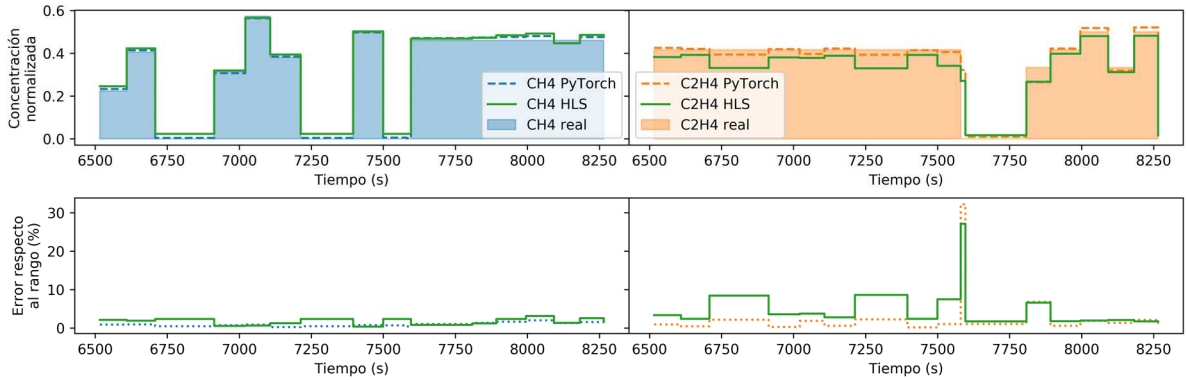



**Figura 21:** Predicciones durante la ventana de ejemplo con la red implementada en HLS junto con los errores asociados.

tres capas, que son básicamente multiplicaciones matriciales, por lo que están compuestas por dos bucles anidados; el código de descripción se recoge en el Anexo VIII. Por otro lado, el código en C/C++ no solo debe contener el modelo, sino que también debe especificar cuáles son los tipos de datos a usar. Para ello se hace uso de la librería de Vivado HLS "*ap_fixed.h*", que permite definir datos con representación en punto fijo de hasta 32 bits, permitiendo además fijar los métodos de cuantización (cuando el valor a representar necesita de una mayor precisión) y de *overflow* (cuando el valor a representar está fuera del rango representable). Además, haremos uso de la librería "*hls_math.h*", también de Vivado HLS, que permite usar funciones matemáticas con datos representados en punto fijo, necesario en nuestro caso para poder implementar la función exponencial dentro de la sigmoide.

Una vez descrito el sistema a implementar en C/C++ teniendo en cuenta los tipos de datos de punto fijo, es necesario escribir un código que sirva de banco de test, capaz de llamar al modelo de C/C++ y comprobar que la salida de su simulación es la esperada. En nuestro caso hemos usado el que aparece en el Anexo IX, donde se cargan los pesos generados por PyTorch y se obtienen las salidas de la red de C/C++ para cada uno de los datos filtrados del problema bajo estudio, comparándolas con las salidas que se obtienen con PyTorch. En la Figura 21 se muestran las predicciones para las concentraciones obtenidas con la red implementada en PyTorch y en HLS. Como vemos, el error aumenta en este segundo caso, ya que en PyTorch únicamente cuantizábamos los pesos, mientras que en HLS se cuantizan todos los datos (pesos, operaciones no lineales y datos de entrada) del modelo, dando lugar, por tanto, a un mayor error en la salida de la red.

### 4.1.2. Síntesis

Tal y como hemos comentado anteriormente, en la síntesis del modelo sobre la FPGA se busca identificar las operaciones que debe realizar el diseño y asignarle a cada una de ellas un recurso y unos ciclos de reloj concretos, dentro de todo el tiempo de operación. Este es un proceso automatizado de HLS, que busca la mejor solución posible dentro de las directrices que se le indican. De esta forma, podemos especificar los requerimientos necesarios para el diseño, como pueden ser el periodo/frecuencia de reloj, interfaces de comunicación para los puertos, técnicas de optimización de tiempo y/o recursos… Así pues, es en este punto donde vamos a tratar de sopesar dos de los objetivos que nos marcábamos al comienzo del trabajo: el tiempo de inferencia y el uso de área, los cuales, como ya mencionamos en su momento, mantienen un compromiso entre sí.

| | Latencia (ciclos) | DSP (%) | FF (%) | LUT (%) |
|---|---|---|---|---|
| **Sin directrices** | 5615 | 5 | 2 | 10 |
| **Unroll** | 3797 | 31 | 6 | 24 |
| **Unroll+Pipeline** | 275 | 31 | 11 | 28 |

**Tabla I:** Resultados de la latencia y el uso del área por recursos: Procesador digital de señal (DSP), biestables (*flipflops*, FF) y tablas de consulta (*look-up tables*, LUT) para las diferentes optimizaciones del diseño durante la síntesis.

Antes de tratar de especificar ninguna directriz sobre el diseño, realizamos una primera síntesis de éste para poder establecer el punto de partida en cuanto a latencia (ciclos de reloj que tarda el diseño en inferir una salida a partir de sus entradas) y uso de recursos (% de uso de cada uno de los tipos de recursos hardware disponibles en la FPGA). Como vemos en la Tabla I, la latencia es de 5615 ciclos de reloj, lo que hace que usando un reloj de periodo de 10 ns, tengamos un tiempo de inferencia de 56,15 µs, 10 veces mayor al que se consigue al inferir usando PyTorch en nuestra CPU que es de 5,75 µs.

Conocido el punto de partida del diseño, el objetivo es optimizarlo para reducir la latencia, a costa de un mayor uso de área y recursos lógicos. Existen diferentes estrategias que permiten alcanzar este objetivo; en nuestro caso nos ha bastado con dos: el *unrolling* y el *pipelining*.

Como ya hemos comentado, nuestro sistema está compuesto por tres bucles exteriores (uno por capa), los cuales tienen cada uno otro interior (Figura 22 (a)). Por tanto, lo primero que hemos hecho para optimizar el diseño ha sido *desenrollar* los bucles interiores, incluyendo una directriz de *unroll*. Esto implica implementar físicamente cada una de las iteraciones del bucle para que se ejecuten de forma paralela (Figura 22 (b)), de forma que la latencia se reduce unos 2000 ciclos de reloj, reduciendo el tiempo de inferencia a 37,97 µs (Tabla I); el consumo de área aumenta significativamente, aunque se mantiene por debajo de un tercio en cada uno de los recursos, por lo que consideramos que es un compromiso aceptable.

Por otro lado, en el bucle interior de nuestro diseño estamos realizando el sumatorio de los pesos ponderados de la neurona (Figura 2 (a)), y a pesar de que los bucles estén desenrollados estos no se pueden dar de forma paralela porque necesitan recibir el valor del acumulador, que es la salida de la iteración anterior. Una forma de solventar este problema es realizar *pipelining*, es decir, anticipar todas las operaciones posibles cuando los recursos

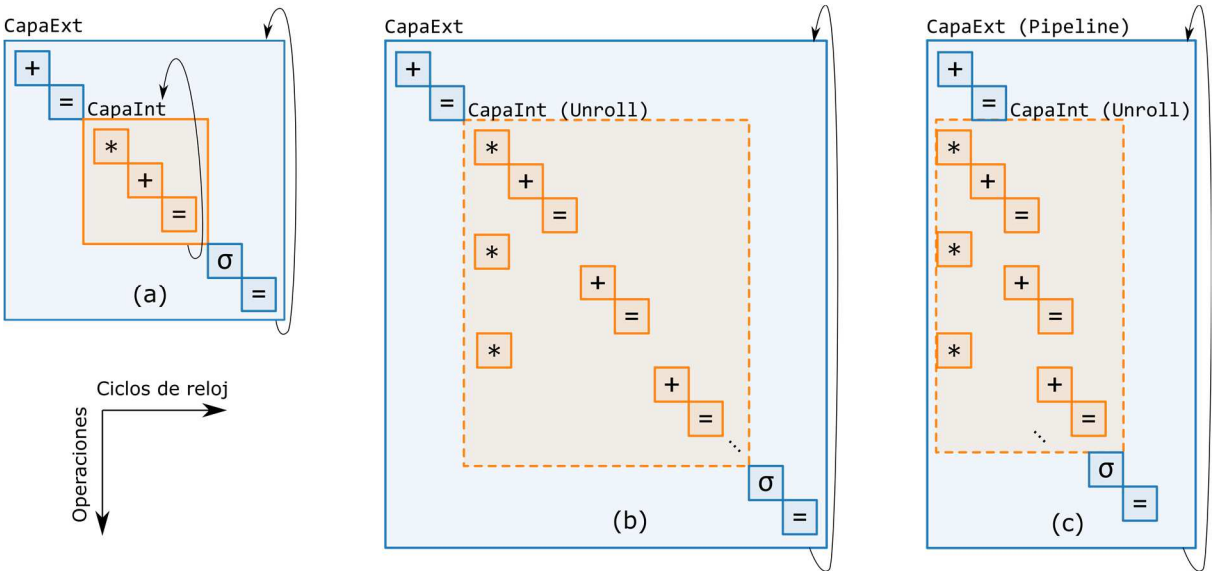



**Figura 22:** Esquema simplificado de la programación de las operaciones a realizar en una capa con tres diferentes aproximaciones: (a) sin directrices, (b) *unroll* del bucle interior y (c) *unroll* del bucle interior y *pipeline* del exterior.

estén libres. Esto implica precargar las entradas y pesos y encadenar operaciones de forma que estas se den de forma simultánea (Figura 22 (c)), lo que reduce significativamente la latencia del sistema a costa de un ligero aumento del uso de área. Ahora bien, esto también necesita de un mayor acceso a la memoria, lo que implica realizar particiones de ésta, aumentando así el ancho de banda necesario. En nuestro caso, al incluir la directriz de *pipeline* en los bucles externos, se consigue reducir el tiempo de inferencia hasta los 2,75 µs, por debajo de la mitad del de la CPU.

Finalmente, queremos valorar el uso de la función sigmoide como función de activación. Si bien su expresión implica recursos aritméticos de gran coste, como son la división y la exponenciación, y una gran latencia (en nuestro caso ha sido de 38 ciclos de reloj), el hecho de que el sistema global esté bajo las directrices de *unroll* y *pipeline*, permite compatibilizar su uso con un tiempo de inferencia reducido, a la vez que se aprovechan las características de derivabilidad y continuidad de esta función de salida, adecuadas para la resolución de problemas de regresión no lineal.

### 4.1.3. Validación

Una vez que Vivado HLS asigna a cada operación unos recursos y un ciclo de reloj durante la síntesis, es necesario realizar una simulación en HDL que verifique que los resultados coinciden con la simulación en C/C++. Para ello genera los archivos de HDL (VHDL o Verilog, a elección), que contienen la descripción del sistema teniendo en cuenta las directrices tomadas durante la síntesis, así como un archivo basado en el banco de test, que permite validar el diseño desde HDL.

El principal problema es el coste computacional cuando el sistema es muy complejo. En nuestro caso, sólo hemos sido capaces de finalizar la simulación sin directrices. En cambio, al realizar el *unroll* y el *pipeline* sobre cada una de las capas, el sistema hace uso de muchos más recursos lógicos y de conectividad dentro del dispositivo programable, lo que hace que la simulación que evalúa su salida de forma concurrente necesite de una gran potencia de cálculo, algo de lo que no disponíamos durante la realización de esta parte del trabajo, y por lo que no hemos sido capaces de verificar el funcionamiento del sistema optimizado con las técnicas de *unroll* y *pipeline*. Por tanto, en lo que sigue de esta memoria vamos a trabajar siempre con el diseño básico, sin optimizar.

## 4.2. Implementación en Vivado

Hasta ahora hemos usado Vivado HLS para diseñar, simular, sintetizar y validar nuestro diseño; una vez llegado este punto es necesario disponer de una herramienta que nos permita generar el archivo *bitstream* encargado de configurar la matriz de conexionado de nuestra FPGA. Para ello vamos a hacer uso del programa Vivado de Xilinx, una herramienta más completa que Vivado HLS (éste sólo está dedicado a HLS), que también está incluido en el paquete WebPack.

Dentro de todas sus funcionalidades, nos interesa el diseñador por bloques, una herramienta que permite un diseño rápido de sistemas basados en bloques IP (*Intellectual Property*), disponibles dentro de un repositorio del propio programa. Uno de estos bloques es el ZYNQ *Processing System*, encargado de embeber las diferentes interfaces presentes en los SoCs

de la familia ZYNQ (de la cual nuestra plataforma formará parte) para poder controlar otros bloques IPs. Además de los bloques presentes en el repositorio, Vivado también permite importar bloques IP diseñados en HLS y es lo que hacemos con nuestra red neuronal una vez que su diseño ha sido validado.

En la Figura 23 aparece el diseño por bloques del sistema que se va a incluir en la FPGA. Además del ZYNQ *Processing System* y el módulo de la red, incluye dos bloques más. El situado a la izquierda es un bloque de *reset*, generado de forma automática por la herramienta Vivado y que sirve para establecer un sistema de reinicio de todo el sistema. El segundo bloque, en mitad de la figura, es un sistema de interconexión de la interfaz AXI (*Advanced eXtensible Interface*). Este tipo de interfaz es parte de la arquitectura de bus de los microcontroladores ARM y es la que nos va a permitir leer y escribir en los bloques de nuestro módulo de red neuronal desde el software del microcontrolador mediante Python.

Una vez que el diseño de bloques está completo, la herramienta Vivado permite realizar una síntesis del sistema completo y una virtualización de la implementación, tras lo cual es posible generar un archivo *bitstream* que cargaremos en nuestra FPGA.

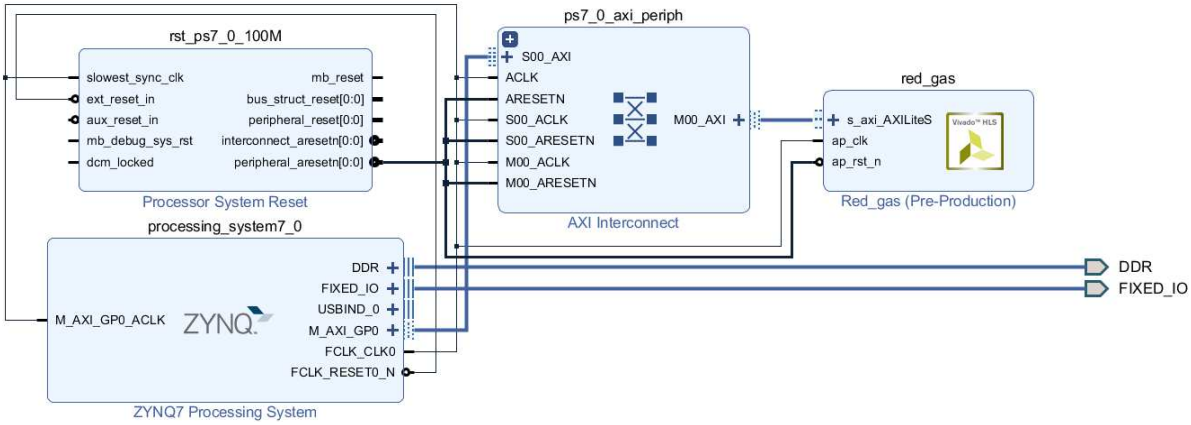



**Figura 23:** Diseño por bloques del sistema final. Destacan el bloque de la red y el ZYNQ *Processing System*.

# 4.3. PYNQ y uso de la red

Antes de continuar con la descripción del uso de la red neuronal sobre la FPGA, presentamos brevemente la plataforma sobre la que vamos a hacerlo. Esta plataforma es PYNQ Z2 [24] [25], basada en una placa que incluye un SoC (*System on Chip*) de Xilinx de la familia Zynq, concretamente el ZYNQ XC7Z020-1CLG400C, que incluye un procesador Cortex-A9 de dos núcleos, una FPGA con 13300 CLBs y 220 DSP *slices* y 512 Mb de memoria DDR3. Sobre este sistema se puede instalar una versión de Linux basada en cuadernos de Jupyter que permite el control tanto del procesador como de la lógica programable contenida en la FPGA mediante Python (Figura 24).

Así pues, la plataforma PYNQ nos va a permitir cargar nuestro diseño en la FPGA y controlarla desde Python. Para ello PYNQ dispone de un paquete específico que cuenta con todas las funciones necesarias, habilitando estas funcionalidades de una forma sencilla de cara al usuario. Aun así, puesto que en nuestro caso hemos usado una representación en punto fijo para los datos y, concretamente, para las salidas y las entradas, vamos a necesitar desarrollar un controlador específico para poder leer y escribir datos en punto fijo desde Python, que usa representación en punto flotante. Este controlador es el que aparece en el Anexo X, donde se embebe tanto la transformación de datos como las direcciones de los puertos de las entradas

y salidas, dejando de cara al usuario dos funciones básicas para el uso del modelo: 'set_params(·)', que permite la carga de los parámetros entrenados a partir de un fichero, y 'pred(·)', la función diseñada para escribir las entradas y leer las salidas de la red.

Una vez desarrollado el controlador obtenemos las salidas de la red neuronal implementada sobre la FPGA (ventana de ejemplo en la Figura 24) y comprobamos que son las que obteníamos con la simulación en Vivado HLS. Este código está en el Anexo XI.

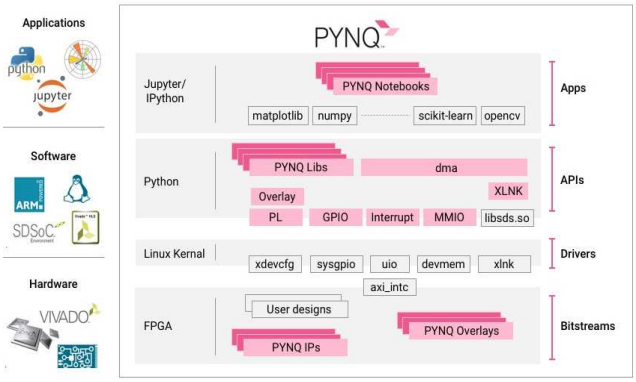



**Figura 24:** Esquema de la plataforma PYNQ, incluyendo las capas hardware, software y de aplicaciones. Fuente [27].

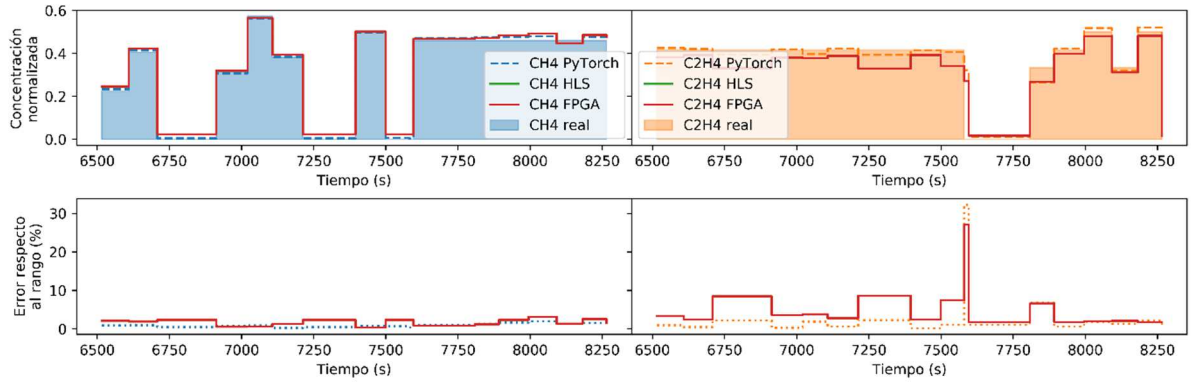



**Figura 25:** Salidas de la red neuronal implementada en la FPGA junto con su error asociado. También se muestran las concentraciones reales, las predecidas por PyTorch y las obtenidas en la simulación de HLS. Nótese que estas últimas están solapadas por las de la FPGA.

# 5. Conclusiones

El flujo de trabajo definido ha permitido cumplir nuestro objetivo principal: la implementación de una red neuronal en una FPGA, capaz de inferir la concentración de dos gases a partir de las salidas de 16 sensores de una nariz artificial. Para ello, una vez procesados los datos de nuestro *dataset*, normalizándolos y realizando un promediado de ventana temporal para recuperar el comportamiento estático de los sensores, hemos diseñado una arquitectura óptima de red neuronal, virtualizada en una CPU para poder entrenarla mediante PyTorch. Este entrenamiento se ha realizado en dos fases: la primera, finalizada cuando la red estaba cercana al punto de aprendizaje final, ha permitido determinar la representación en punto fijo para los datos de la red más adecuada para alcanzar un error acotado; en la segunda, hemos introducido la cuantificación inherente a dicho tipo de dato, obteniendo unos parámetros del modelo cuantificados, en los que la representación escogida no afecta al error final de estimación. A continuación, en la fase de implantación de la red hemos completado su descripción en HLS, resultando un bloque IP importable desde el diseñador por bloques de Vivado, que ha permitido generar un *bitstream* para configurar adecuadamente la matriz de conexionado físico en la FPGA. Finalmente hemos desarrollado un controlador que integra todo lo necesario para trabajar con la red neuronal implementada en la FPGA desde la plataforma PYNQ, usando Python. La validación de este flujo de trabajo se ha realizado con un segundo *dataset* [20] [21], cuyos resultados se presentan en el Anexo XII.

Por otro lado, en las primeras fases del proyecto nos planteamos tres objetivos específicos para validar la red: la reducción del tiempo de inferencia, la elección de una representación de dato oportuna y un uso de área eficiente, objetivos que competían entre sí, y creemos que hemos conseguido una solución que consigue aunar estos tres aspectos. En primer lugar, el tipo de dato seleccionado es de punto fijo en complemento a 2 con 12 bits totales, de los cuales 3 se utilizan para la parte entera. El hecho de que sea de punto fijo reduce la complejidad de la aritmética y la selección del número de bits está suficientemente justificada con los resultados de la Figura 16, donde se muestra como el tipo de dato que mejor compromiso error/número de bits alcanza. Por su parte, el uso de área es bastante contenido a pesar de las directrices incluidas durante la síntesis del diseño, ya que, en la versión optimizada (*unroll* y *pipeline*), el uso de los tres tipos de recursos (LUTs, FFs y DSPs) era inferior al 33%. Y, finalmente, el tiempo de inferencia en dicha versión optimizada es muy reducido, inferior a la mitad del conseguido en CPU, lo que es sorprendente para un sistema tan relativamente sencillo como el que hemos presentado.

Aunque nos damos por satisfechos con lo conseguido con el trabajo queremos dedicar estas líneas finales al posible trabajo futuro que se abre con la finalización de este proyecto. El hecho de no sólo implementar una red neuronal en una FPGA sino de haber definido todo un flujo de trabajo nos abre la puerta a su extrapolación tanto con otro tipo de redes neuronales (CNNs, *Convolutional Neural Networks*, de gran interés en visión artificial; LSTM, *Long Sort-Term Memory*, un tipo especial de redes neuronales recurrentes con un marcado uso en procesamiento del lenguaje natural o NLP…), como con otros *datasets* que permitan la fusión sensorial, como pueden ser la clasificación en calidad alimentaria a partir de lecturas de sensores bacterianos, o la determinación del nivel de maduración de productos frescos en función de niveles de VoCs específicos en su atmósfera. En definitiva, creemos que este trabajo nos puede servir como punto de partida para otras aplicaciones de redes neuronales implementadas sobre FPGAs.

# Bibliografía


[1] S. Haykin, Nerual Networks, Macmillan College Publishing Company, 1994.

[2] N. J. Medrano Marqués y B. Martin del Brio, «Sensor linearization with neural networks,» *IEEE Transactions on Industrial Electronics,* vol. 48, nº 6, pp. 1288 - 1290, 2001.

[3] A. Jain, A. Singh, H. S. Koppula, S. S. y A. Saxena, «Recurrent Neural Networks for driver activity anticipation via sensory-fusion architecture,» de *IEEE International Conference on Robotics and Automation (ICRA)*, Stockholm, 2016.

[4] D. A. Dornfeld y M. F. DeVries, «Neural Network Sensor Fusion for Tool Condition Monitoring,,» *CIRP Annals,* vol. 39, pp. 101-105, 1990.

[5] Z. Chen, Y. Zheng, K. Chen y J. Jian, «Concentration Estimator of Mixed VOC Gases Using Sensor Array With Neural Networks and Decision Tree Learning,» *IEEE Sensors Journal,* vol. 17, nº 6, pp. 1884-1892, 2017.

[6] S. L. Bade y B. L. Hutchings, «FPGA-based stochastic neural networks-implementation,» de *Proceedings of IEEE Workshop on FPGA's for Custom Computing Machines*, Napa Valley, CA, USA, 1994.

[7] R. Gadea, J. Cerdá, F. Ballester y A. Mocholí, «Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation,» de *Proceedings of the 13th international symposium on System synthesis (ISSS)*, Madrid, España, 2000.

[8] GitHub Inc., «The State of the Octoverse,» 2019. [En línea]. Available: https://octoverse.github.com/. [Último acceso: 30 03 20].

[9] N. A. Woods y T. VanCourt, «FPGA acceleration of quasi-Monte Carlo in finance,» de *International Conference on Field Programmable Logic and Applications*, Heidelberg, 2008.

[10] E. B. Fernandez, W. A. Najjar, S. Lonardi y J. Villarreal, «Multithreaded FPGA acceleration of DNA sequence mapping,» de *IEEE Conference on High Performance Extreme Computing*, Waltham, 2012.

[11] D. Rumelhart, G. Hinton y R. Williams, «Learning representations by back-propagating errors,» *Nature ,* vol. 323, pp. 533-536, 1986.

[12] G. Cybenko, «Approximation by Superpositions of a Sigmoidal Function,» *Mathematics of Control, Signals and Systems,* vol. 2, pp. 303-314, 1989.

[13] M. Leshno, L. Y. Vladimir, P. Allan y S. Shimon, «Multilayer feedforward networks with a nonpolynomial activation function can approximate any function,,» *Neural Networks,* vol. 6, nº 6, pp. 861-867, 1993.

[14] M. I. Masud, FPGA Routing Structures: A Novel Switch Block and Depopulated Interconnect Matrix Architectures, The University of British Columbia, 1998.

[15] L. Torres y Quevedo, «Ensayos sobre Automática,» *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales,* vol. XII, pp. 319-419, 1913.

[16] D. Lin, S. Talathi y S. Annapureddy, «Fixed point quantization of deep convolutional networks.,» de *International Conference on Machine Learning*, New York, 2016.

[17] A. Zhou, A. Yao, Y. Guo, L. Xu y Y. Chen, «Incremental network quantization: Towards lossless cnns with low-precision weights,» *arXiv preprint,* 2017.

[18] A. Srivastava, «Detection of volatile organic compounds (VOCs) using SnO2 gas-sensor array and artificial neural network,» *Sensors and Actuators B: Chemical,* vol. 96, nº 1-2, pp. 24-37, 2003.

[19] M. Penza y G. Cassano, «Application of principal component analysis and artificial neural networks to recognize the individual VOCs of methanol/2-propanol in a binary mixture by SAW multi-sensor array,» *Sensors and Actuators B: Chemical,* vol. 89, nº 3, pp. 269-284, 2003.

[20] J. Fonollosa, S. Sheik, R. Huerta y S. Marco, «Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring,» *Sensors and Actuators B: Chemical,* vol. 215, pp. 618-629, 2015.

[21] D. Dua y C. Graff, «UCI Machine Learning Repository,» University of California, School of Information and Computer Science, 2020. [En línea]. Available: http://archive.ics.uci.edu/ml.

[22] «Figaro Engineering Inc.,» [En línea]. Available: https://www.figaro.co.jp/en/.

[23] D. P. Kingma y J. L. Ba, «Adam: A Method for Stochastic Optimization,» de *International Conference on Learning Representations (ICLR)*, San Diego (CA, EE.UU.), 2015.

[24] Xilinx, «PYNQ: Python Productivity,» [En línea]. Available: http://www.pynq.io/home.html.

[25] TUL, «PYNQ Z2 Product Specification,» [En línea]. Available: http://www.tul.com.tw/ProductsPYNQ-Z2.html.

[26] B. Bhattarai, «FPGA Implementation of CORDIC Processor».

[27] Edge AI, «Xilinx,» [En línea]. Available: https://www.xilinx.com/applications/industrial/analytics-machine-learning.html.

[28] «VHDL.es,» [En línea]. Available: https://vhdl.es/fpga/.

# Lista de acrónimos

- ALU: *Arithmetic Logic Unit*, Unidad Aritmeticológica
- ANN: *Artificial Neural Network*, Red Neuronal Artificial
- ASIC: *Application-Specific Integrated Circuit,* Circuito Integrado de Aplicación Especifica
- AXI: *Advanced eXtensible Interface*, Interfaz Extensible Avanzada
- CNN: *Convolutional Neural Network*, Red Neuronal Convolucional
- CPU: *Central Processing Unit*, Unidad Central de Procesamiento
- DSP: *Digital Signal Processing,* Procesado de Señal Digital
- FPGA: *Field-Programmable Gate Array*, Matriz de Puertas lógicas Programables en Campo
- GPU: *Graphics Processing Unit*, Unidad de Procesado de gráficos
- HDL: *Hardware Description Language*, Lenguaje de Descripción de Hardware
- HLS: *High-Level Synthesis*, Síntesis de Alto Nivel
- IP: *Intellectual Property,* Propiedad Intelectual
- LSTM: *Long Sort-Term Memory*, Memoria Larga de Corta Duración
- LUT: *LookUp Table,* Tabla de Consulta
- ML: *Machine Learning,* Aprendizaje Máquina
- MSE: *Mean Square Error*, Error Cuadrático Medio
- NLP: *Natural Language Processing,* Procesado Natural del Lenguaje
- PLL: *Phase-Locked Loop*, Bucles de Enganche de Fase
- RAM: *Random Access Memory*, Memoria de Acceso Aleatorio
- RTL: *Register-Transfer Level*, Nivel de transferencia de Registros
- SGD: *Stochastic Gradient Descent*, Descenso de Gradientes Estocástico.
- SoC: *System on Chip*, Sistema en Chip
- VHDL: VHSIC-HDL: *Very High Speed Integrated Circuit Hardware Description Languange,* Lenguaje de Descripción de Hardware de Circuitos Integrados de Alta Velocidad
- VOC: *Volatile Organic Compound*, Compuesto Orgánico Volátil

# Lista de figuras

# Anexos

Los anexos de I a VII y de X a XII se corresponden a cuadernos de Jupyter, por lo que trataremos de mostrarlos lo más parecidos a éste.

## I. Preparación del *dataset*

En este anexo vamos a presentar el código que filtra los datos para llegar a la Figura 15. En primer lugar, cargamos las librerías, los datos y les echamos un vistazo.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display, clear_output
import time

np.random.seed(1)

t, CH4, Eth = np.loadtxt('data/ethylene_methane.txt', skiprows=1, usecols=
(0,1,2), unpack=True)
sens = np.loadtxt('data/ethylene_methane.txt', skiprows=1, usecols=(3,4,5,
6,7,8,9,10,11,12,13,14,15,16,17,18))

orig = 6350
fin = orig+2000

fig, ax1 = plt.subplots(figsize=(12.0,4.0), dpi=600)

ax2 = ax1.twinx()

ax2.set_ylabel('Concentración (ppm)')
ax2.fill_between(t[np.logical_and(orig<t, t<fin)], CH4[np.logical_and(orig
<t, t<fin)], alpha=0.4, label='CH4')
ax2.fill_between(t[np.logical_and(orig<t, t<fin)], Eth[np.logical_and(orig
<t, t<fin)], alpha=0.4, label='Etileno')

ax1.set_xlabel('Tiempo (s)')
ax1.set_ylabel('Señal sensores')
for i in range(16):
    ax1.plot(t[np.logical_and(orig<t, t<fin)], sens[np.logical_and(orig<t,
 t<fin),i], label='Canal '+str(i))
ax1.legend(loc='center left', bbox_to_anchor=(1.08, 0.5))

ax2.legend(loc='upper right')

fig.tight_layout()

fig.savefig('ver-datos.png')

fig.tight_layout()
```

*Aquí obtendríamos la Figura 12.*

Una vez llego a este punto el siguiente paso es normalizar.

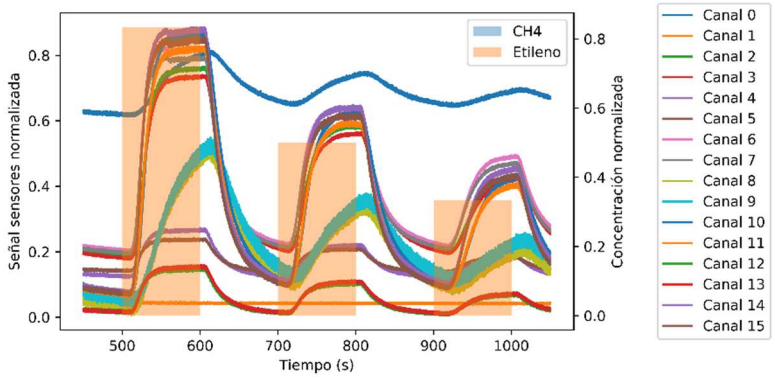
```
CH4max = CH4.max(); CH4min = CH4.min();
Ethmax = Eth.max(); Ethmin = Eth.min();
CH4norm = (CH4 - CH4min)/(CH4max - CH4min)
Ethnorm = (Eth - Ethmin)/(Ethmax - Ethmin)
sensnorm = np.empty(sens.shape)
for i in range(16):
    sensnorm[:,i] = (sens[:,i] - sens[:,i].min())/(sens[:,i].max() - sens[
:,i].min())

orig = 6350
fin = orig+2000

fig, ax1 = plt.subplots(figsize=(12.0,4.0), dpi=600)

ax2 = ax1.twinx()

ax2.set_ylabel('Concentración normalizada')
ax2.fill_between(t[np.logical_and(orig<t, t<fin)], CH4norm[np.logical_and(
orig<t, t<fin)], alpha=0.4, label='CH4')
ax2.fill_between(t[np.logical_and(orig<t, t<fin)], Ethnorm[np.logical_and(
orig<t, t<fin)], alpha=0.4, label='Etileno')

ax1.set_xlabel('Tiempo (s)')
ax1.set_ylabel('Señal sensores normalizada')
for i in range(16):
    ax1.plot(t[np.logical_and(orig<t, t<fin)], sensnorm[np.logical_and(ori
g<t, t<fin),i], label='Canal '+str(i))
ax1.legend(loc='center left', bbox_to_anchor=(1.08, 0.5))

ax2.legend(loc='upper right')

fig.tight_layout()
fig.savefig('ver-datos-normalizados.png')
```

*Aquí obtenemos la Figura 13.*

Puesto que queremos emular un comportamiento 'estático' de los sensores químicos de gas, primero debemos filtrar los datos de manera que nos quedemos con un valor que indique la tendencia de la señal del sensor. Pero antes debemos alinear los cambios en concentración con los cambios en la señal de los sensores. Para hacer esto vamos a mover todos los datos de la señal de los sensores $N_{sinc}$ (`Nsinc` en el código) posiciones hacia la izquierda. Este será un parámetro del procesado.

```
Nsinc = -1000 # El - es porque es hacia la izquierda

senssinc = np.roll(sensnorm, Nsinc, axis=0)

orig = 450
fin = 1050

fig, ax1 = plt.subplots(figsize=(8.0,4.0))
```

```
ax2 = ax1.twinx()

ax2.set_ylabel('Concentración normalizada')
ax2.fill_between(t[np.logical_and(orig<t, t<fin)], CH4norm[np.logical_and(
orig<t, t<fin)], alpha=0.4, label='CH4')
ax2.fill_between(t[np.logical_and(orig<t, t<fin)], Ethnorm[np.logical_and(
orig<t, t<fin)], alpha=0.4, label='Etileno')

ax1.set_xlabel('Tiempo (s)')
ax1.set_ylabel('Señal sensores normalizada')
for i in range(16):
    ax1.plot(t[np.logical_and(orig<t, t<fin)], senssinc[np.logical_and(ori
g<t, t<fin),i], label='Canal '+str(i))
ax1.legend(loc='center left', bbox_to_anchor=(1.15, 0.5))

ax2.legend(loc='upper right')

fig.tight_layout()
```




Como vemos, más o menos los cambios en la salida de los sensores ya se corresponden con los de concentración. Para poder hacer el último paso, el promediado de la parte estacionaria, primero vamos a identificar los cambios de concentración de gas, que serán lo que llamemos '`flancos`'. A la vez generamos también los vectores donde se almacenarán los datos del tiempo y las concentraciones de los nuevos datos. Hay que tener en cuenta que vamos a pasar de un montón de datos (~1e4) para cada estado de concentraciones a sólo uno, necesitamos nuevos vectores para guardar el tiempo y las concentraciones normalizadas del Etileno y el $CH_4$.

```
Nflancos = 0
Ndatos = len(t)
tNuevo = []
PositFlancos = []
CH4normNuevo = []
EthnormNuevo = []


for i in range(Ndatos): #Primero barremos los tiempos
    if(CH4norm[i]!=CH4norm[i-1] or Ethnorm[i]!=Ethnorm[i-1]):
        Nflancos += 1
        tNuevo.append(t[i])
        PositFlancos.append(i)
```

```
        CH4normNuevo.append(CH4norm[i])
        EthnormNuevo.append(Ethnorm[i])
```

Ahora vamos a calcular el tiempo entre un flanco y su anterior y promediaremos el `FraccPromediado%` de los datos de cada sensor que estén pegados al flanco de la izquierda.

```
FraccPromedidado = 0.1
sensEstat = np.empty([Nflancos,16])
for i in range(1,Nflancos):
    for j in range(16):
        sensEstat[i-1,j] = np.mean(senssinc[(PositFlancos[i-1]+int((1-Frac
cPromedidado)*(PositFlancos[i]-PositFlancos[i-1]))):PositFlancos[i],j])

tNuevo = np.array(tNuevo)
CH4normNuevo = np.array(CH4normNuevo)
EthnormNuevo = np.array(EthnormNuevo)
```

De esta forma hemos conseguido un comportamiento estático de los sensores para las concentraciones de Etileno y $CH_4$. Veamos cómo queda representando los sensores en el mismo rango temporal que en la anterior figura.

```
orig = 6350
fin = orig+2000

fig, ax1 = plt.subplots(figsize=(12.0,4.0), dpi=600)


ax2 = ax1.twinx()

ax2.set_ylabel('Concentración normalizada')
ax2.fill_between(tNuevo[np.logical_and(orig<tNuevo, tNuevo<fin)], CH4normN
uevo[np.logical_and(orig<tNuevo, tNuevo<fin)], step='post', alpha=0.4, lab
el='CH4')
ax2.fill_between(tNuevo[np.logical_and(orig<tNuevo, tNuevo<fin)], EthnormN
uevo[np.logical_and(orig<tNuevo, tNuevo<fin)], step='post', alpha=0.4, lab
el='Etileno')

ax1.set_xlabel('Tiempo (s)')
ax1.set_ylabel('Señal de salida de los sensores')
for i in range(16):
    ax1.step(tNuevo[np.logical_and(orig<tNuevo, tNuevo<fin)], sensEstat[np
.logical_and(orig<tNuevo, tNuevo<fin),i], where='post', label='Canal '+str
(i))
ax1.legend(loc='center left', bbox_to_anchor=(1.08, 0.5))

ax2.legend(loc='upper right')

fig.tight_layout()

fig.savefig('ver-datos-filtrados.png')
```

*Aquí obtenemos la Figura 15.*

Guardamos estos datos. Quitamos los dos últimos valores porque con `np.roll` se han visto afectados y han perdido el sentido.

```
np.savetxt('filtered-data/ethylene_methane.txt', np.column_stack([tNuevo[:
len(tNuevo)-2], CH4normNuevo[:len(tNuevo)-2], EthnormNuevo[:len(tNuevo)-2]
, sensEstat[:len(tNuevo)-2,:]]), header='Time(s) CH4(norm) Eth(norm) filte
red sensor readings (16 channels)')
t =tNuevo[:len(tNuevo)-2]
CH4 = CH4normNuevo[:len(tNuevo)-2]
Eth = EthnormNuevo[:len(tNuevo)-2]
sens = sensEstat[:len(tNuevo)-2,:]
```

# II. *Datasets* en PyTorch

En este anexo vamos a preparar los datos filtrados para poder usarlos en PyTorch. Para ello primero debemos separar el *dataset* que acabamos de filtrar en dos, uno para el *training set* y otro para el *validation set*, de modo que vamos a cortar en dos la *time series* que tenemos. Con `TS_VS_factor`=(num. datos training set)/(num. datos totales) = 1-(# datos validation set)/(# datos totales) definimos dónde está éste corte.

```
TS_VS_factor = 0.7
tTS, tVS = np.split(t, [int(TS_VS_factor*len(t))])
CH4TS, CH4VS = np.split(CH4, [int(TS_VS_factor*len(t))])
EthTS, EthVS = np.split(Eth, [int(TS_VS_factor*len(t))])
sensTS, sensVS = np.split(sens, [int(TS_VS_factor*len(t))])
```

Ahora configuramos los datos para crear el *training set* (`train_ds` en el código) y el *validation set* (`valid_ds` en el código).

```
gasTS = np.stack([CH4TS, EthTS])
gasTS = np.transpose(gasTS)

gasVS = np.stack([CH4VS, EthVS])
gasVS = np.transpose(gasVS)
```

Echemos un vistazo a la dimensión de los siguientes objetos.

```
gasTS.shape, sensTS.shape
```

```
((244, 2), (244, 16))
```

A continuación, los pasamos a tensores y creamos los objetos *training set* y *validation set*.

```
sensTS = torch.tensor(sensTS, dtype=torch.float)
gasTS = torch.tensor(gasTS, dtype=torch.float)
train_ds = torch.utils.data.TensorDataset(sensTS, gasTS)

sensVS = torch.tensor(sensVS, dtype=torch.float)
gasVS = torch.tensor(gasVS, dtype=torch.float)
valid_ds = torch.utils.data.TensorDataset(sensVS, gasVS)
```

Creamos los cargadores de datos, *data loaders*. Para ello tenemos que definir unos *batch sizes* que son los datos que va a procesar de golpe la red. Esto nos permite cargar sólo esos en memoria. Ahora no es muy relevante, pero en caso de tener mucha más cantidad de datos es interesante tenerlo en cuenta. En el `train_dl` al colocar el argumento `shuffle` como `True` le estamos diciendo al *data loader* que reordene los datos aleatoriamente en cada entrenamiento.

```
bsTS = gasTS.shape[0]//2
bsVS = gasVS.shape[0]//1

train_dl = torch.utils.data.DataLoader(train_ds, batch_size=bsTS, shuffle=
True)
valid_dl = torch.utils.data.DataLoader(valid_ds, batch_size=bsVS)
```

Una vez creados los *data loaders* ya tenemos los datos listos para entrenar la red.

# III. Definición de la arquitectura de red

El propósito de este anexo es mostrar el código para poder definir la red. En este caso vamos a usar una red *feed fordward* de cuatro capas (dos ocultas) con el siguiente número de neuronas por cada capa:

```
D_in, H1, H2, D_out = 16, 32, 12, 2
```

Usando estos parámetros definimos la red y la inicializamos:

```
class TwoHiddenLayerNet(torch.nn.Module):
    def __init__(self, D_in, H1, H2, D_out):
        """
        En el 'constructor' inicializamos las capas dandoles
        las dimensiones de entrada y salida de cada una de ellas
        y las asignamos como variables de la clase
        """
        super(TwoHiddenLayerNet, self).__init__()
        self.linear1 = nn.Linear(D_in, H1)
        self.linear2 = nn.Linear(H1, H2)
        self.linear3 = nn.Linear(H2, D_out)

    def forward(self, x):
        """
        En la función de propagación, 'forward', recogemos el
        tensor de entrada y configuramos como es su propagación
        hasta la salida. Podemos usar las capas definidas en el
        'constructor' o realizar cualquier operación sobre él.
        A la salida hay función de activación porque queremos
        obtener las cocnetraciones de gas.
        """
        x = self.linear1(x)
        x = F.sigmoid(x)
        x = self.linear2(x)
        x = F.sigmoid(x)
        x = self.linear3(x)
        x = F.sigmoid(x)
        return x

# Inicializamos el modelo
model = TwoHiddenLayerNet(D_in, H1, H2, D_out)
```

# IV. Entrenamiento de la red

Este anexo contiene el código para entrenar la red. Primero es necesario definir el número de épocas totales a entrenar, el *learning rate*, la función de coste (`criterion`) y el algoritmo de entrenamiento (`optimizer`).

```
epochs, learning_rate = 1e4, 1e-4

criterion = torch.nn.MSELoss(reduction='sum')
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Entrenamos la red hasta un número máximo de épocas.

```
t0 = time.time()

train_loss_data =[]
train_dl_len = len(train_dl)
valid_loss_data =[]
valid_dl_len = len(valid_dl)
for epoch in range(int(epochs)+1):
    model.train()
    for xb, yb in train_dl:
        pred = model(xb)
        loss = criterion(pred, yb)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    model.eval()
    with torch.no_grad():
        train_loss = sum(criterion(model(xb), yb) for xb, yb in train_dl)/
train_dl_len
        train_loss_data.append(train_loss)
        valid_loss = sum(criterion(model(xb), yb) for xb, yb in valid_dl)/
valid_dl_len
        valid_loss_data.append(valid_loss)

        if epoch%100==0:
            clear_output(wait=True)
            display('Epoch: '+str(epoch)+', Train Loss: '+'{:05.3f}'.forma
t(train_loss)+', Valid Loss: '+'{:05.3f}'.format(valid_loss) + ', Min Vali
d Loss: '+'{:05.3f}'.format(min(valid_loss_data)))

    last_epoch = epoch

print('Tiempo de entrenamiento: '+str(time.time()-t0))
```

```
Epoch: 10000, Train Loss: 0.459, Valid Loss: 0.885, Min Valid Loss: 0.885
Tiempo de entrenamiento: 133.1488814353943
```

Veamos cómo han evolucionado las métricos durante el entrenamiento.

```
orig = 6400
fin = orig + 1000

fig, axs = plt.subplots(ncols=2, figsize=(16,4))

axs[0].semilogy(np.arange(last_epoch+1), train_loss_data, label='Training
set')
axs[0].semilogy(np.arange(last_epoch+1), valid_loss_data, label= 'Validati
on set')
```

```
#axs[0].set_title('Evolución del entrenamiento')
axs[0].set_ylabel('Coste')
axs[0].set_xlabel('Época');
axs[0].legend()


with torch.no_grad():
    axs[1].fill_between(t[np.logical_and(orig<t, t<fin)], CH4[np.logical_a
nd(orig<t, t<fin)], alpha=0.4, step='post', label='CH4 real')
    axs[1].fill_between(t[np.logical_and(orig<t, t<fin)], Eth[np.logical_a
nd(orig<t, t<fin)], alpha=0.4, step='post', label='Eth real')

    axs[1].step(t[np.logical_and(orig<t, t<fin)], CH4pred[np.logical_and(o
rig<t, t<fin)], 'C0--', where='post', label='CH4 pred')
    axs[1].step(t[np.logical_and(orig<t, t<fin)], Ethpred[np.logical_and(o
rig<t, t<fin)], 'C1--', where='post', label='Eth pred')

    axs[1].legend(loc='lower right')
    axs[1].set_xlabel('Tiempo (s)')
    axs[1].set_ylabel('Concentración normalizada')

    plt.savefig('ver-resultados.png', dpi=600)
```

*Aquí va la Figura 16.*

Guardamos el modelo

```
torch.save(model.state_dict(), './EthCH4_VL'+str(valid_loss)+'_model.pth')
```

# V. Entrenamiento cuantizado de la red

Para tener en cuenta el paso a punto fijo de la red, vamos a cuantizar los pesos. Para ello vamos a emular un paso a punto fijo mediante una función.

```
def ToFixedTensor(TensorOrig, NbitsTot, NbitsInt):
    """
    Esta función cuantiza el dato original a los datos representables por
    un punto fijo de NbitsTot bits de ancho total y NbitsInt de ancho en
    el entero. Emula el método de HLS usando overflow AP_SAT y cuanti-
    zación AP_RND
    """
    if len(TensorOrig.size())==2:
        rows, cols = TensorOrig.size()

        FixedTensor = torch.empty_like(TensorOrig)

        NbitsFrac = NbitsTot - NbitsInt

        for i in range(rows):
            for j in range(cols):
                FracOrig, IntOrig = modf(TensorOrig[i,j])

                IntOrig = int(IntOrig)

                # Estudiamos el overflow
```

```
                MaxInt = 2**(NbitsInt - 1) - 1 #Definimos los límites
                MinInt = -2**(NbitsInt - 1)

                if IntOrig > MaxInt:
                    IntFixed = MaxInt
                elif IntOrig < MinInt:
                    IntFixed = MinInt
                else: IntFixed = IntOrig

                #Estudiamos la cuantización
                deltaFrac = 1/2**NbitsFrac #Definimos la delta de cuantiza
ción

                FracFixed = deltaFrac*round(FracOrig/deltaFrac)

                #Devolvemos el punto fijo
                FixedTensor[i,j] =  IntFixed + FracFixed

    elif len(TensorOrig.size())==1:
        cols = TensorOrig.size()[0]

        FixedTensor = torch.empty_like(TensorOrig)

        NbitsFrac = NbitsTot - NbitsInt

        for i in range(cols):
            FracOrig, IntOrig = modf(TensorOrig[i])

            IntOrig = int(IntOrig)

            # Estudiamos el overflow
            MaxInt = 2**(NbitsInt - 1) - 1 #Definimos los límites
            MinInt = -2**(NbitsInt - 1)

            if IntOrig > MaxInt:
                IntFixed = MaxInt
            elif IntOrig < MinInt:
                IntFixed = MinInt
            else: IntFixed = IntOrig

            #Estudiamos la cuantización
            deltaFrac = 1/2**NbitsFrac #Definimos la delta de cuantización

            FracFixed = deltaFrac*round(FracOrig/deltaFrac)

            #Devolvemos el punto fijo
            FixedTensor[i] =  IntFixed + FracFixed
    else: raise ValueError('Tensor de dimensión superior a dos')
    return FixedTensor
```

```
def ToFixed(Orig, NbitsTot, NbitsInt):
    """
```

```
    Esta función cuantiza el dato original a los datos represntables por
    un punto fijo de NbitsTot bits de ancho total y NbitsInt de ancho en
    el entero. Emula el método de HLS usando overflow AP_SAT y cuanti-
    zación AP_RND
    """

    NbitsFrac = NbitsTot - NbitsInt

    FracOrig, IntOrig = modf(Orig)

    IntOrig = int(IntOrig)

    # Estudiamos el overflow
    MaxInt = 2**(NbitsInt - 1) - 1 #Definimos los límites
    MinInt = -2**(NbitsInt - 1)

    if IntOrig > MaxInt:
        IntFixed = MaxInt
    elif IntOrig < MinInt:
        IntFixed = MinInt
    else: IntFixed = IntOrig

    #Estudiamos la cuantización
    deltaFrac = 1/2**NbitsFrac #Definimos la delta de cuantización

    FracFixed = deltaFrac*round(FracOrig/deltaFrac)

    #Devolvemos el punto fijo
    return IntFixed + FracFixed
```

Entrenamos la red hasta un número máximo de épocas

```
epochs = 5e3

t0 = time.time()

train_loss_data =[]
train_dl_len = len(train_dl)
valid_loss_data =[]
valid_dl_len = len(valid_dl)

model.load_state_dict(torch.load('./EthCH4_VLtensor(0.8446)_model.pth'))

for epoch in range(int(epochs)+1):
    model.train()
    for xb, yb in train_dl:
        pred = model(xb)
        loss = criterion(pred, yb)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    model.eval()
```

```
    with torch.no_grad():
        train_loss = sum(criterion(model(xb), yb) for xb, yb in train_dl)/
train_dl_len
        train_loss_data.append(train_loss)
        valid_loss = sum(criterion(model(xb), yb) for xb, yb in valid_dl)/
valid_dl_len
        valid_loss_data.append(valid_loss)

        if epoch%100==0:
            clear_output(wait=True)
            display('Epoch: '+str(epoch)+', Train Loss: '+'{:05.3f}'.forma
t(train_loss)+', Valid Loss: '+'{:05.3f}'.format(valid_loss) + ', Min Vali
d Loss: '+'{:05.3f}'.format(min(valid_loss_data)))

        if epoch % 100 == 0: #Cada 100 iteracciones hacemos:
            for param in model.parameters():
                param.data = ToFixedTensor(param.data, 12, 3)

    last_epoch = epoch

print('Tiempo de entrenamiento: '+str(time.time()-t0))
```

```
'Epoch: 5000, Train Loss: 0.280, Valid Loss: 0.801, Min Valid Loss: 0.800'
Tiempo de entrenamiento: 65.90973806381226
```

Veamos cómo han evolucionado las métricos durante el entrenamiento.

```
plt.semilogy(np.arange(last_epoch+1), train_loss_data, label='Training set
')
plt.semilogy(np.arange(last_epoch+1), valid_loss_data, label= 'Validation
set')
plt.title('Evolución del entrenamiento')
plt.ylabel('Coste')
plt.xlabel('Época');
plt.legend()

plt.savefig('EthCH4_Entrenaiento_VL'+str(valid_loss)+'.png', dpi=600)
```

*Aquí va la figura 18.*

# VI. Análisis de los resultados del entrenamiento

En esta parte vamos a tratar de cuantificar cómo de bien funciona la red. Para ello vamos a computar el error obtenido para cada par de combinaciones de concentraciones.

```
errCH4 = np.abs(((CH4pred.detach().numpy()-CH4)*(CH4max-
CH4min)+CH4min)/(CH4max-CH4min))*100
errEth = np.abs(((Ethpred.detach().numpy()-Eth)*(Ethmax-
Ethmin)+Ethmin)/(Ethmax-Ethmin))*100

errTot = np.sqrt(errCH4**2 + errEth**2)
```

```
orig = 6400
fin = orig + 1000

with torch.no_grad():
```

```
    fig, axs = plt.subplots(nrows=2, ncols=2,  sharey='row', figsize=(10,5
))

    axs[0,0].fill_between(t[np.logical_and(orig<t, t<fin)], CH4[np.logical
_and(orig<t, t<fin)], color='C0', alpha=0.4, step='post', label='CH4 real'
)
    axs[0,1].fill_between(t[np.logical_and(orig<t, t<fin)], Eth[np.logical
_and(orig<t, t<fin)], color='C1', alpha=0.4, step='post', label='Eth real'
)

    axs[0,0].step(t[np.logical_and(orig<t, t<fin)], CH4pred[np.logical_and
(orig<t, t<fin)], 'C0--', where='post', label='CH4 pred')
    axs[0,1].step(t[np.logical_and(orig<t, t<fin)], Ethpred[np.logical_and
(orig<t, t<fin)], 'C1--', where='post', label='Eth pred')

    #ax2.step(t[np.logical_and(orig<t, t<fin)], errTot[np.logical_and(orig
<t, t<fin)], 'gray', where='post', label='Error total')
    axs[1,0].step(t[np.logical_and(orig<t, t<fin)], errCH4[np.logical_and(
orig<t, t<fin)], 'C0:', where='post', label='Error CH4')
    axs[1,1].step(t[np.logical_and(orig<t, t<fin)], errEth[np.logical_and(
orig<t, t<fin)], 'C1:', where='post', label='Error Eth')


    for i in range(2):
        for j in range(2): axs[i,j].set_xlabel('Tiempo (s)')
    axs[0,0].set_ylabel('Concentración normalizada')
    axs[1,0].set_ylabel('Error respecto al rango (%)')

    axs[0,0].legend(loc='lower left')
    axs[0,1].legend(loc='lower left')

    fig.subplots_adjust(wspace=0, hspace=0.4)
```

*Aquí iria la figura 19.*

```
errCH4ord = np.sort(errCH4)
errEthord = np.sort(errEth)

terrMaxCH4 = t[np.where(errCH4==errCH4ord[-1])]
terrMaxEth = t[np.where(errEth==errEthord[-1])]

dcha = 500
izqda = 440

with torch.no_grad():
    fig, axs = plt.subplots(nrows=2, sharex='col', figsize=(10,8))

    axs[0].fill_between(t[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+
dcha)], CH4[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)], color=
'C0', alpha=0.4, step='post', label='CH4 real')
    axs[0].fill_between(t[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+
dcha)], Eth[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)], color=
'C1', alpha=0.4, step='post', label='Eth real')
```

```
    axs[0].step(t[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)],
CH4pred[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)], 'C0--', wh
ere='post', label='CH4 pred')
    axs[0].step(t[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+anc)], E
thpred[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)], 'C1--', whe
re='post', label='Eth pred')

    axs[1].step(t[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)],
errCH4[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)], 'C0:', wher
e='post', label='Error CH4')
    axs[1].step(t[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)],
errEth[np.logical_and(terrMaxCH4-izqda<t, t<terrMaxCH4+dcha)], 'C1:', wher
e='post', label='Error Eth')

    for i in range(2):
        axs[i].margins(-0.4,0.05)
    axs[0].set_ylabel('Concentración normalizada')
    axs[1].set_ylabel('Error respecto al rango (%)')
    axs[1].set_xlabel('Tiempo (s)')
    axs[0].legend(loc='upper left')
    axs[1].legend(loc='upper left')

    fig.subplots_adjust(wspace=0, hspace=0)


    #fig.savefig('peor-error.png', dpi=600)
```

*Aquí iria la Figura 21.*

```
CH4errsort=[]
Etherrsort=[]
for i in range(len(t)):
    CH4errsort.append(CH4[np.where(errCH4==errCH4ord[i])])
    Etherrsort.append(Eth[np.where(errEth==errEthord[i])])
```

```
plt.axhline(y=10, label='Cambio de tipo de error', c='k', ls='--')
plt.plot([0.4, 1],[2e-3, 5], 'C3--', label='Tendencia error minimo')

plt.semilogy(CH4errsort, errCH4ord, '+', label='CH4')
plt.semilogy(Etherrsort, errEthord, '+', label='Eth')

plt.title('Errores')
plt.ylabel('Error respecto al rango (%)')
plt.xlabel('Concentración normalizada')
plt.legend()

plt.savefig('errores.png')
```

*Aquí iría la Figura 20.*

# VII. Exportación de datos para C++

En este anexo presentamos el anexo para el guardado de los pesos y la generación de un fichero para poder comprobar el funcionamiento de la red usadno C++.

```
fout = open("eth-ch4-params-quant.txt","w")
for name, param in model.named_parameters():
    np.savetxt(fout, param.detach().numpy(), fmt='%.6f')
    #print(name, param)
fout.close()
```

```
orig = 0
fin = orig+100000
with torch.no_grad():
    np.savetxt('test-cpp/ethylene_methane_qunat_'+str(orig)+'-'+str(fin)+'
.txt', np.column_stack([t[np.logical_and(orig<t, t<fin)],
    CH4[np.logical_and(orig<t, t<fin)],
    Eth[np.logical_and(orig<t, t<fin)],
    CH4pred[np.logical_and(orig<t, t<fin)],
    Ethpred[np.logical_and(orig<t, t<fin)],
    sens[np.logical_and(orig<t, t<fin),:]]),
    header='Time(s) CH4r(norm) Ethr(norm) CH4p(norm) Ethp(norm) filtered s
ensor readings (16 channels)')
```

# VIII. Código en C/C++ de la red neuronal

Este es el archivo donde se encuentra el modelo.

```
#include "red-gas.h"

x_data_t sigmoid(x_data_t x);

x_data_t sigmoid(x_data_t x){
        return 1/(1+hls::expf(-x)); //Sigmoid
}

void red_gas(x_data_t in[D_in],
             w_data_t w01[D_in][H1],  b_data_t b01[H1],
             w_data_t w12[H1][H2],    b_data_t b12[H2],
             w_data_t w23[H2][D_out], b_data_t b23[D_out],
             x_data_t out[D_out]){

    int i,j;

    x_data_t tmp, inter1[H1], inter2[H2];

    Capa01Ext: for(i=0;i<H1;i++){
        tmp = b01[i];
        Capa01Int: for(j=0;j<D_in;j++){
            tmp += w01[j][i]*in[j];
        }
        inter1[i] = sigmoid(tmp);
    }

    Capa12Ext: for(i=0;i<H2;i++){
        tmp = b12[i];
        Capa12Int: for(j=0;j<H1;j++){
            tmp += w12[j][i]*inter1[j];
```

```
        }
        inter2[i] = sigmoid(tmp);
    }

    Capa23Ext: for(i=0;i<D_out;i++){
        tmp = b23[i];
        Capa23Int: for(j=0;j<H2;j++){
            tmp += w23[j][i]*inter2[j];
        }
        out[i] = sigmoid(tmp);
    }
}
```

Aquí está la librería donde se inicializan las constantes y se asignan los tipos de datos. (‘red_gas.h’).

```
#include "hls_math.h"
#include "ap_fixed.h"

using namespace std;

#define D_in 16
#define H1 32
#define H2 12
#define D_out 2

typedef ap_fixed<12,3,AP_RND,AP_SAT> b_data_t;
typedef ap_fixed<12,3,AP_RND,AP_SAT> x_data_t;
typedef ap_fixed<12,3,AP_RND,AP_SAT> w_data_t;

void red_gas(x_data_t in[D_in],
             w_data_t w01[D_in][H1],  b_data_t b01[H1],
             w_data_t w12[H1][H2],    b_data_t b12[H2],
             w_data_t w23[H2][D_out], b_data_t b23[D_out],
             x_data_t out[D_out]);
```

# IX. Banco de test de Vivado HLS

```
#include "red-gas.h"
#include <stdio.h>
#include <iostream>

int main(){

    x_data_t out[D_out], in[D_in];
    w_data_t w01[D_in][H1],  w12[H1][H2], w23[H2][D_out];
    b_data_t b01[H1], b12[H2], b23[D_out];

    int i, j, k;
    float t, out_real[D_out], out_py[D_out], tmp;

    cout << "Inicio programa\nLeemos los pesos...";
```

```
    FILE *fparam;
    fparam =
fopen("C:/Users/eneri/OneDrive/Universidad/Master/TFM/Virtualizacion/Red-
Gas/eth-ch4-params-quant.txt","r");
    if(fparam == NULL){
        cout << "Error al abrir el archivo de parametros" << endl;
        return 0;
    }

    //Leemos w01
    for(i=0;i<H1;i++){
        for(j=0;j<D_in;j++){
            fscanf(fparam,"%f ",&tmp);
            w01[j][i] = tmp;
        }
        fscanf(fparam,"\n");
    }

    //Leemos b01
    for(i=0;i<H1;i++){
        fscanf(fparam,"%f\n",&tmp);
        b01[i] = tmp;
    }

    //Leemos w12
    for(i=0;i<H2;i++){
        for(j=0;j<H1;j++){
            fscanf(fparam,"%f ",&tmp);
            w12[j][i] = tmp;
        }
        fscanf(fparam,"\n");
    }

    //Leemos b12
    for(i=0;i<H2;i++){
        fscanf(fparam,"%f\n",&tmp);
        b12[i] = tmp;
    }

    //Leemos w23
    for(i=0;i<D_out;i++){
        for(j=0;j<H2;j++){
            fscanf(fparam,"%f ",&tmp);
            w23[j][i] = tmp;
        }
        fscanf(fparam,"\n");
    }

    //Leemos b23
    for(i=0;i<D_out;i++){
        fscanf(fparam,"%f\n",&tmp);
        b23[i] = tmp;
    }
```

```
    fclose(fparam);


    cout << "OK\nAbriendo fichero de test...";

    FILE *fin;
    fin =
fopen("C:/Users/eneri/OneDrive/Universidad/Master/TFM/Virtualizacion/Red-
Gas/test-cpp/ethylene_methane_quant_0-100000.txt", "r");
    if(fin == NULL){
        cout << "Error al abrir el archivo de test" << endl;
        return 0;
    }

    fscanf(fin,"# Time(s) CH4r(norm) Ethr(norm) CH4p(norm) Ethp(norm)
filtered sensor readings (16 channels)\n");

    cout << "OK\nAbriendo fichero de salida...";

    FILE *fout;
    fout = fopen("out_ethylene_methane.txt", "w");
    if(fout == NULL){
        cout << "Error al abrir el archivo de salida" << endl;
        return 0;
    }

    fprintf(fout, "Temp(s) CH4real Ethreal CH4py Ethpy CH4pp Ethcpp\n");

    cout << "OK\nComenzando el test\n";

    k=1;

    while(k<300){
        cout << "\tLinea " << k << "...";

        fscanf(fin, "%f %f %f %f %f ", &t,
                                       &out_real[0],
                                       &out_real[1],
                                       &out_py[0],
                                       &out_py[1]);
        for(i=0;i<D_in;i++){
               fscanf(fin,"%f ", &tmp);
               in[i] = tmp;
        }
        fscanf(fin, "\n");


        red_gas(in, w01, b01, w12, b12, w23, b23, out);

        fprintf(fout, "%f %f %f %f %f %f %f\n", t,
                                                out_real[0],
                                                out_real[1],
                                                out_py[0],
                                                out_py[1],
```

```
                                                (float)out[0],
                                                (float)out[1]);

        cout << "OK\n";
        k++;
    }

    return system("FC output_golden.txt out_ethylene_methane.txt");
}
```

# X. Driver para el control del bloque de la red neuronal desde Python

```
import numpy as np
import matplotlib.pyplot as plt
from pynq import Overlay, DefaultIP
from bitstring import Bits
from time import sleep

class RedGas(DefaultIP):
    def __init__(self,description):
        super().__init__(description=description)

    def set_params(self, path_pesos):

        """
        Permite escribir los pesos en la FPGA, se debe proporcionar el
        directorio de un archivo .npz donde los pesos estén ordenados
        de la siguiente manera: 'w01', 'b01', 'w12', 'b12', 'w23', 'b23'
        """
        w01 = np.load(path_pesos)['w01']
        b01 = np.load(path_pesos)['b01']
        w12 = np.load(path_pesos)['w12']
        b12 = np.load(path_pesos)['b12']
        w23 = np.load(path_pesos)['w23']
        b23 = np.load(path_pesos)['b23']

        #Comenzamos con el peso w01
        self.write_array(w01.flatten('F'), 0x0400)
        #Lo mimso con b01 y posteriores. Los b ya están en fila.
        self.write_array(b01, 0x0800)

        self.write_array(w12.flatten('F'), 0x0c00)
        self.write_array(b12, 0x1000)

        self.write_array(w23.flatten('F'), 0x1040)
        self.write_array(b23, 0x1080)

    def pred(self, entrada):

        """
        Permite inferir un resultado a partir de una entrada,
```

```
        que debe ser un array con las salidas normalizadas de
        los senosores
        """

        self.write_array(entrada, 0x0020)

        sleep(1e-3)

        out = Bits(int=self.read(0x1088), length=32)
        eth=(out[4:16]).int/(2**9)
        ch4=(out[20:]).int/(2**9)

        return ch4, eth

    def write_array(self, inputArray, offset):
        """
        Recibe un array y un offset inicial y escribe los datos de
        ese array partiendo desde la dirección del offset usando el
        formato:
        Memory 'inputArray' (512 * 12b)
                            Word n : bit [11: 0] - inputArray[2n]
                                     bit [27:16] - inputArray[2n+1]
                                     others      - reserved
        """
        for i in range(int(len(inputArray)/2)): #Cada dos pesos

            # Calculamos el entero correspondiente según lo que obtenemos
en el fichero._hw.h
            try:
                self.write(offset,('0b0000'+Bits(int=int(round(inputArray[
2*i+1]*(2**9))), length=12)+
                                           '0b0000'+Bits(int=int(round(inputArray[
2*i]*(2**9))), length=12)).int)
            except:
                print(inputArray[2*i], inputArray[2*i+1])
                print(round(inputArray[2*i]*(2**9)), round(inputArray[2*i+
1]*(2**9)))
                raise

            offset+=4


    bindto = [overlay.ip_dict['red_gas']['type']] #Esto debe sacar 'xilinx
.com:hls:red_gas:1.0'
```

# XI. Ejemplo de uso de la red desde Python

Es necesario cargar la *overlay* e inicializar el modelo.

```
overlay = Overlay('/home/xilinx/pynq/overlays/red_gas/red_gas.bit')
modelo = overlay.red_gas
modelo.set_params('eth-ch4-params-quant.npz')
```

Cargamos los datos

```
t, CH4, Eth = np.loadtxt('ethylene_methane.txt', skiprows=1, usecols=(0,1,
2), unpack=True)
sens = np.loadtxt('ethylene_methane.txt', skiprows=1, usecols=(3,4,5,6,7,8
,9,10,11,12,13,14,15,16,17,18))
```

Probamos a predecir

```
elto=250
modelo.pred(sens[elto,:]), CH4[elto], Eth[elto]
```

```
((0.25, 0.017578125), 0.23595240502915696, 0.0)
```

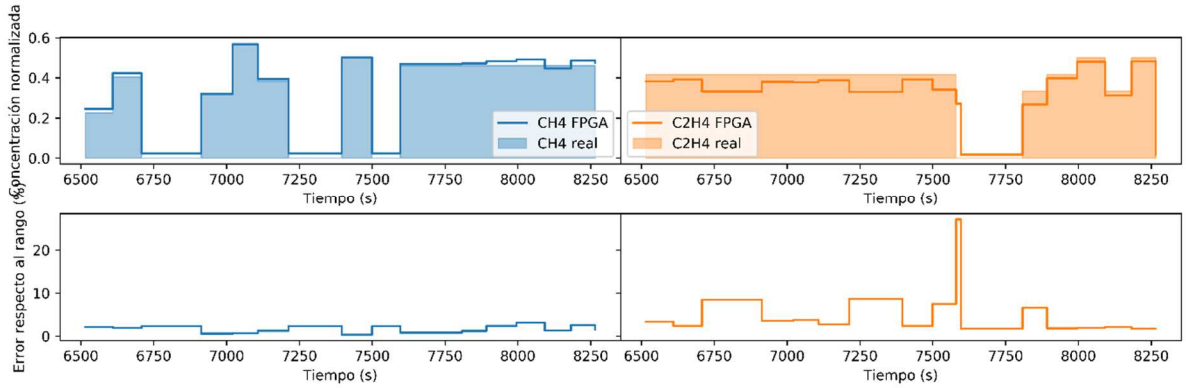
Funciona. Vamos a sacar toda la time series

```
CH4FPGA = np.empty_like(CH4)
EthFPGA = np.empty_like(Eth)
for i in range(len(t)):
    (CH4FPGA[i], EthFPGA[i]) = modelo.pred(sens[i, :])

CH4max, CH4min, Ethmax, Ethmin = 296.67, 0.0, 20.0, 0.0

errCH4 = np.abs(((CH4FPGA-CH4)*(CH4max-CH4min)+CH4min)/(CH4max-CH4min))*10
0
errEth = np.abs(((EthFPGA-Eth)*(Ethmax-Ethmin)+Ethmin)/(Ethmax-Ethmin))*10
0
```

Representamos.

```
orig = 6350
fin = orig + 2000

fig, axs = plt.subplots(nrows=2, ncols=2,  sharey='row', figsize=(12.0,4.0
))

axs[0,0].fill_between(t[np.logical_and(orig<t, t<fin)], CH4[np.logical_and
(orig<t, t<fin)], color='C0', alpha=0.4, step='post', label='CH4 real')
axs[0,1].fill_between(t[np.logical_and(orig<t, t<fin)], Eth[np.logical_and
(orig<t, t<fin)], color='C1', alpha=0.4, step='post', label='C2H4 real')

axs[0,0].step(t[np.logical_and(orig<t, t<fin)], CH4FPGA[np.logical_and(ori
g<t, t<fin)], 'C0--', where='post', label='CH4 FPGA')
axs[0,1].step(t[np.logical_and(orig<t, t<fin)], EthFPGA[np.logical_and(ori
g<t, t<fin)], 'C1--', where='post', label='C2H4 FPGA')

#ax2.step(t[np.logical_and(orig<t, t<fin)], errTot[np.logical_and(orig<t,
t<fin)], 'gray', where='post', label='Error total')
axs[1,0].step(t[np.logical_and(orig<t, t<fin)], errCH4[np.logical_and(orig
<t, t<fin)], 'C0:', where='post', label='Error CH4')
axs[1,1].step(t[np.logical_and(orig<t, t<fin)], errEth[np.logical_and(orig
<t, t<fin)], 'C1:', where='post', label='Error C2H4')


for i in range(2):
    for j in range(2): axs[i,j].set_xlabel('Tiempo (s)')
axs[0,0].set_ylabel('Concentración normalizada')
```

```
axs[1,0].set_ylabel('Error respecto al rango (%)')

axs[0,0].legend(loc='lower right')
axs[0,1].legend(loc='lower left')

plt.tight_layout()
fig.subplots_adjust(wspace=0, hspace=0.4)

plt.savefig("time-series-FPGA-12-3-"+str(orig)+"-"+str(fin)+".png",dpi=600
)
```

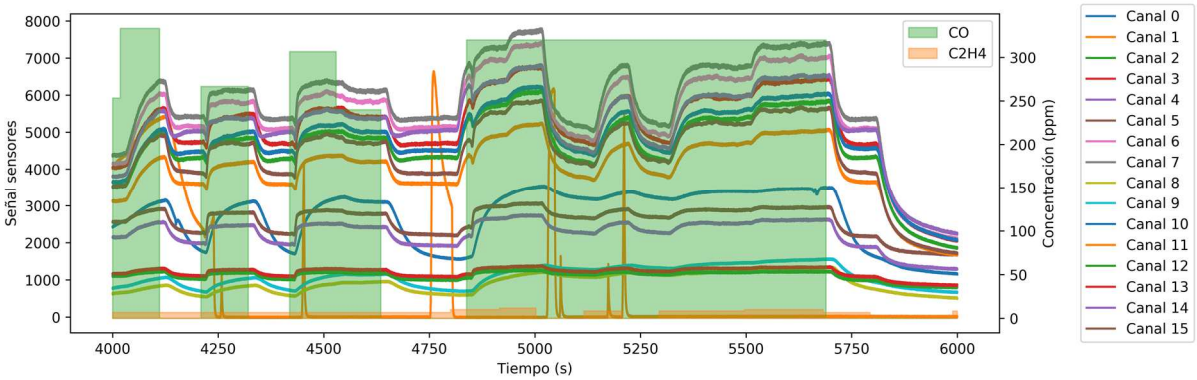



Guardamos los datos

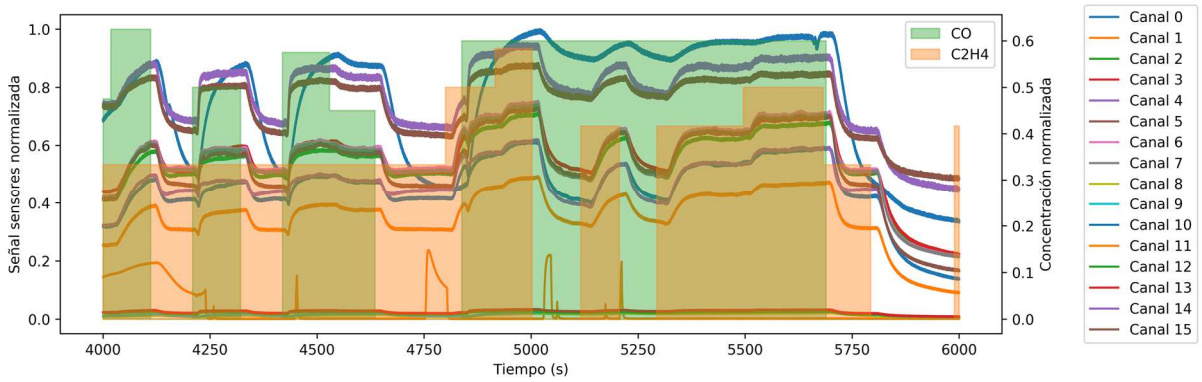
```
np.savez('time-series-fpga.npz', CH4FPGA=CH4FPGA, EthFPGA=EthFPGA)
```

# XII. Validación del flujo de trabajo

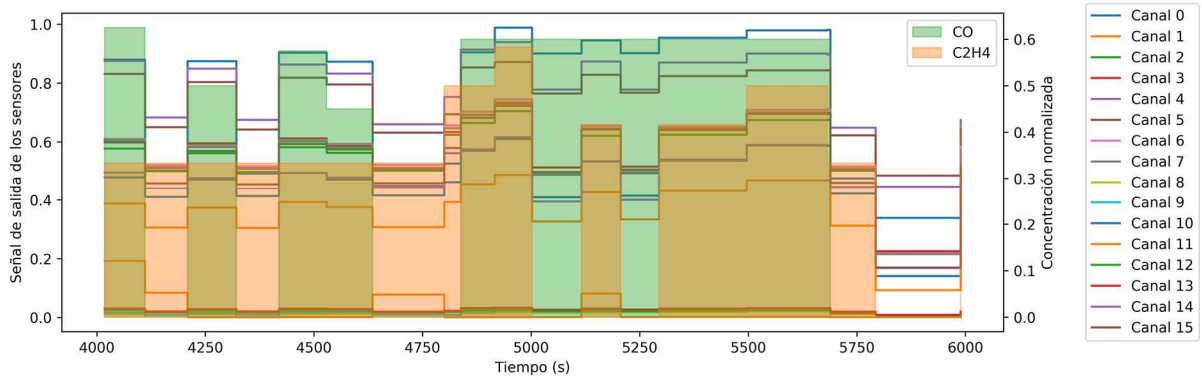
En este anexo nos proponemos dar los mismos pasos descritos en la parte principal de la memoria para validar el flujo de trabajo seguido durante este proyecto. Para ello vamos a partir del otro *dataset* que se incluía en [20] [21], con datos sobre la mezcla de etileno ($C_2H_4$) y monóxido de carbono (CO). Puesto que el código es análogo a lo presentado hasta ahora, sólo vamos a presentar las figuras.



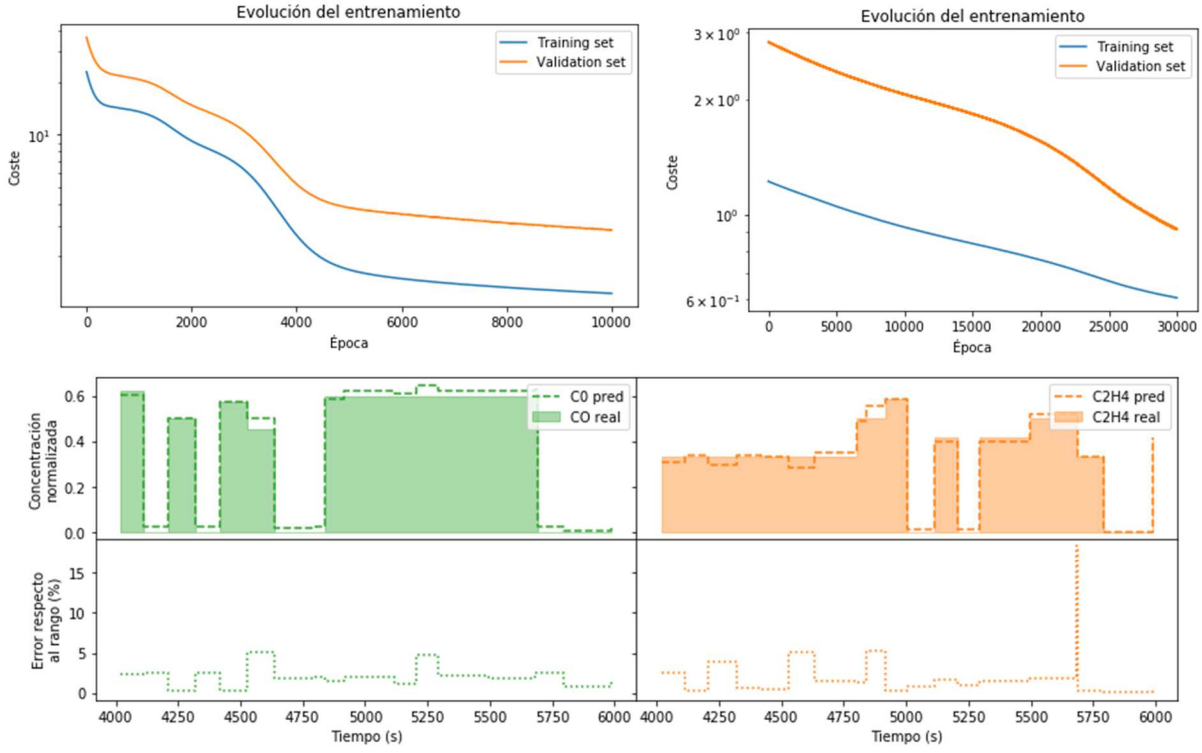

Como vemos, el aspecto que tenemos al echar un vistazo a los datos es similar a lo que conseguíamos en la Figura 12, de forma que comenzamos realizando una normalización.



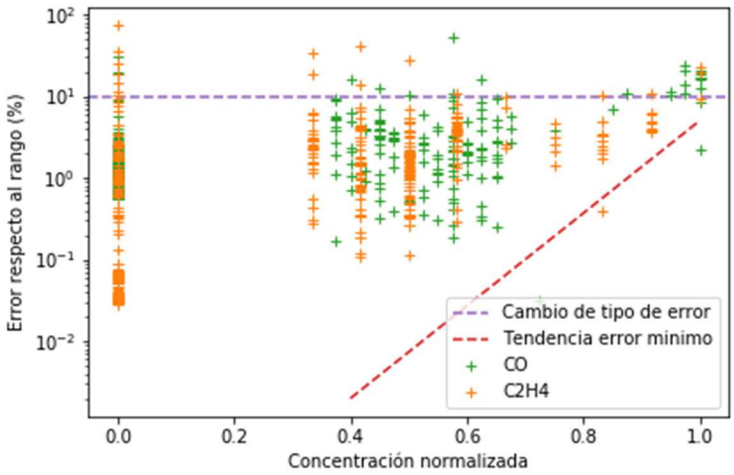

Y a continuación un promediado de ventana temporal que nos permita recuperar el comportamiento estático.



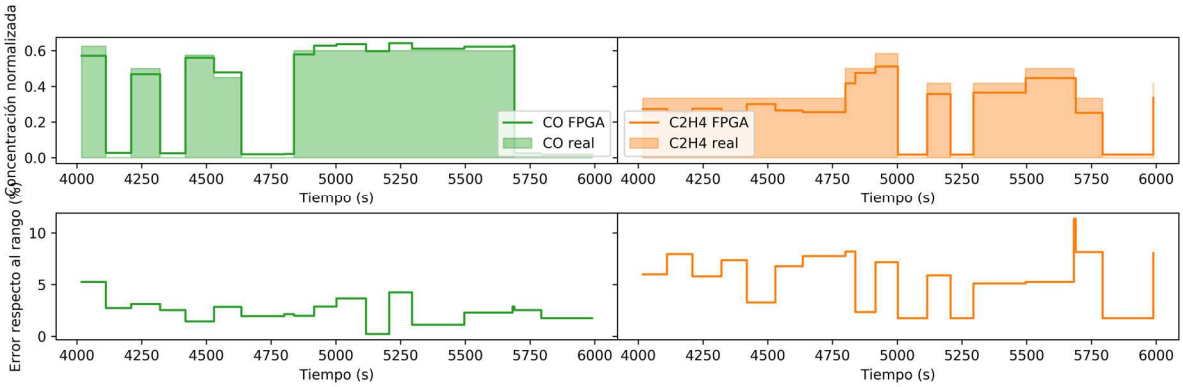

Una vez llegados a este punto ya podemos usar estos datos para entrenar nuestra red neuronal, que va a mantener la arquitectura, puesto que el problema que queremos resolver es el mismo, pero con otros datos. Además, al igual que en el *dataset* anterior vamos a realizar un entrenamiento en dos fases: una primera sin cuantizar y una segunda teniendo en cuenta la cuantización debido a la representación en punto fijo.



Como vemos, la evolución del entrenamiento durante las dos fases y los resultados de las predicciones de la red tienen la misma pinta que para el anterior *dataset*. Para hacer un análisis más riguroso de los errores de la red dibujamos el *plot* de los errores que aparece en la Figura 20 (a).




El tipo de dato para la representación en punto fijo es el mismo, 12-3, ya que el problema es totalmente análogo y la arquitectura de red es la misma. Esto hace que el *bitstream* generado para el anterior problema sea totalmente valido para este nuevo *dataset*, por lo que bastaría cargar los pesos del modelo entrenado y obtener los resultados de toda la serie temporal.




Como vemos los resultados son bastante satisfactorios. Si que es cierto que los errores son ligeramente superiores, pero creemos que estos se podrían reducir algo más alargando el entrenamiento del modelo, que en este caso no ha llegado a estabilizar los indicadores (coste del *training set* y del *validation set* en las figuras de la evolución del entrenamiento) a pesar de que el entrenamiento ha sido más largo, llegando hasta las $4 \cdot 10^4$ épocas.