Diagnóstico de fallas mecánicas en motores eléctricos mediante aplicaciones de inteligencia artificial

Estudiante: Pedro Ivan Wozniak Lorice
(pwozniaklorice@frba.utn.edu.ar)

Director: Ing. Silvio Tapino
(stapino@gmail.com)

Tutor: Ing. Alejandro Furfaro
(afurfaro@electron.frba.utn.edu.ar)

Co-tutor: Ing. Luciano Ferreyro
(lucianopabloferreyro@gmail.com)

Tutores por la cátedra: Dr. Ing. Matias Hampel, Ing. Claudia Orlandi, Ing. Fernando Daniel Fiamberti
(mahampel@frba.utn.edu.ar, orlandiutn@yahoo.com.ar, fernandodaniel35@yahoo.com.ar)

Universidad Tecnológica Nacional
Facultad Regional de Buenos Aires
Departamento de Electrónica

Índice general

Índice de figuras

Índice de tablas

1 Introducción

El número de máquinas eléctricas utilizadas en la industria, procesos de fabricación o generación de energía entre otros es cada vez más grande. En este contexto, es necesario poder mantener un nivel constante de funcionamiento de las máquinas para poder reducir tiempos muertos y ahorrar costos. Con el avance de la industria y la tecnología se buscan nuevas maneras de mejorar la eficiencia de las máquinas así como también la forma de poder predecir el estado de las mismas. El mantenimiento predictivo es de vital importancia para garantizar que las máquinas se mantengan en funcionamiento óptimo. Anticipar y evitar potenciales fallas en un equipo minimiza el riesgo de que una máquina quede fuera de servicio por una falla. Los rodamientos son un componente esencial de cualquier máquina eléctrica. La función de estos elementos es proporcionar un sistema de deslizamiento del rotor dentro del estator manteniendo un espacio de aire uniforme. A través del mantenimiento predictivo es posible detectar las primeras etapas de desarrollo de anomalías, necesarias para poder corregirlas. Una de las herramientas más utilizadas en la actualidad es la detección de anomalías basándonos en el análisis de las vibraciones generadas por la máquina eléctrica.

Este proyecto de grado se enfoca en el análisis de vibraciones, principalmente en el contexto del mantenimiento predictivo, con foco en la detección de fallas tempranas en rodamientos así como también en distintas fallas de una máquina eléctrica, utilizando el dominio del tiempo como técnica de análisis de la señal y luego la utilización de un acelerador de hardware para su procesamiento y clasificación/análisis.

Primero se desarrollarán conceptos básicos para entender el procedimiento en la detección de fallas de rodamientos o máquinas eléctricas, enfocado principalmente en la etapa de detección, ya que ciertos aspectos en el análisis de fallas son subjetivas al conocimiento de la persona que esté realizando el análisis.

En la industria existen muchas alternativas de hardware para procesar estas señales, como lo son los ASICS, GPU y FPGA que compiten por acelerar la inferencia de redes neuronales y mejorar la precisión de las mismas. La flexibilidad, bajo consumo, velocidad y costos de adquisición son sus principales características, especialmente en contextos académicos.

En este proyecto se presentan las bases teóricas y conceptuales del aprendizaje profundo para luego poder abordar el flujo de trabajo implementado para el entrenamiento y validación de la red neuronal diseñada. A sí mismo, se realiza un análisis de impacto de distintas configuraciones de redes neuronales y su influencia en los recursos disponibles de hardware para su procesamiento. En su desarrollo se busca demostrar la implementación de una red neuronal para realizar un diagnóstico de fallas mecánicas en motores eléctricos mediante redes neuronales. Además se busca demostrar la implementación de dicha red neuronal en una FPGA y las limitaciones que poseen este tipo de placas para implementar arquitecturas de redes neuronales así como las restricciones necesarias en el uso de recursos para redes de gran tamaño.

La investigación fue llevada a cabo en el marco de la materia Proyecto Final de la carrera de Ingeniería Electrónica de la Universidad Tecnológica Nacional (Regional Buenos Aires), y fue auspiciada por el Laboratorio de Procesamiento Digital (DPLAB) del Departamento de Electrónica de dicha facultad regional. La investigación realizada y el diseño final quedaron a disposición del DPLAB como base para futuros trabajos de investigación.

2 Materiales y Métodos

2.1. Introducción

Hoy en día, el diagnóstico de fallas sigue siendo un requisito esencial en todas las plantas industriales del mundo. Ha crecido considerablemente en los últimos años, principalmente debido a la aparición de sistemas cada vez más complejos, con el consiguiente aumento del número de procesos a controlar. Debido a los mismos cambios tecnológicos, en respuesta a estas demandas, han surgido diferentes técnicas para la detección e identificación de fallas como solución a problemas que puedan perjudicar el correcto funcionamiento del proceso.

Una planta como un sistema completo, en el que intervienen diferentes tipos de variables a medir y procesos complejos, requiere características como alta calidad y excelente desempeño de sus equipos como principal objetivo para garantizar el cumplimiento de importantes parámetros como la seguridad y la confiabilidad. La confiabilidad se entiende como la capacidad de un componente para realizar la función deseada bajo condiciones dadas dentro de un intervalo de tiempo dado. La confiabilidad se puede expresar en términos de disponibilidad, entendida como una medida del estado del equipo en un momento dado en el que necesita operar.

Uno de los parámetros que contribuye a la disponibilidad del equipo es el mantenimiento del mismo. El mantenimiento considera el impacto de la falla del equipo y cómo minimizarlo. Las diferentes técnicas de mantenimiento generalmente se enfocan en la confiabilidad, utilizando métodos analíticos objetivos, sistemáticos y documentados; centrándose en el mantenimiento preventivo de las partes críticas del equipo y el mantenimiento correctivo de las partes no críticas del equipo.

Pero el problema surge cuando lo que se busca es garantizar la competitividad de la empresa, asegurar la adecuada fiabilidad y disponibilidad de los equipos, respetar las exigencias de calidad y seguridad, que solo pueden garantizarse en el hecho de que todos los bienes materiales estén ininterrumpidos para realizar la programación. Es por esto que, si bien el mantenimiento preventivo se basa en medidas encaminadas a mantener los equipos en buen estado de funcionamiento y evitar fallas, el hecho de que deban realizarse intervenciones periódicas para reemplazar y/o reparar componentes se convierte en su principal desventaja. Es imposible garantizar el mejor nivel de confiabilidad y, al mismo tiempo, genera costos adicionales para las piezas de repuesto que aún son adecuadas para la reparación, incluidos repuestos y materiales, lubricantes, mano de obra y, en su mayoría, equipos de tiempo no disponibles.

Ahora, cuando se trata de fallas en las máquinas, tenemos que hablar de vibración, que a menudo se considera un efecto negativo de un proceso útil. En términos generales, las vibraciones en las máquinas son malas porque provocan desgaste, grietas por fatiga, fallas en los sellos, grietas en el aislamiento, ruido, etc. Al mismo tiempo, la vibración es el mejor indicador del estado mecánico de la máquina y puede predecir con mucha sensibilidad la evolución de los defectos.

Los rodamientos son piezas importantes en la mayoría de las máquinas porque se utilizan para permitir el movimiento relativo entre dos componentes. Los dispositivos en sí constan de cuatro elementos: el anillo interior, el anillo exterior, la jaula y los elementos rodantes. Muchas fallas de la máquina son causadas por los rodamientos, que pueden ser causados por una selección

incorrecta de lubricantes, una instalación incorrecta y más.

2.2. Generalidades

En el mantenimiento predictivo encontramos técnicas que permiten evaluar el estado de las máquinas durante su funcionamiento, como es el caso del análisis de vibraciones, que se define como: técnicas de mantenimiento predictivo que permiten encontrar e identificar las causas de posibles fallas para predecir el problema.

La señal de vibración se mide en función del rango, en función del tiempo o la frecuencia, lo que se conoce como tiempo logarítmico y espectro, respectivamente. Este se denomina espacio de representación y constituye la evolución de la señal bajo una determinada variable espacial (tiempo, frecuencia, frecuencia temporal).

De toda la cadena de procesamiento, la etapa de clasificación es, por lejos, una de las más importantes. Esta etapa es implementada en el último paso de dicha cadena y posee la tarea de, valga la redundancia, clasificar la falla para cada tipo de señal que se adquiera. Actualmente existe una gran cantidad de algoritmos de clasificación, de entre los cuales los más populares en este proceso se encuentran las redes neuronales (como perceptrón multicapa [3], redes neuronales probabilísticas [3], redes ART [3], etc.), máquinas de vectores de soporte (SVM) [4], sistemas difusos [5], métodos estadísticos [6], incluidos clasificadores bayesianos simples [6] y discriminadores lineales [6], métodos basados en la distancia como las distancias euclidianas [6], los árboles de decisión [3], los algoritmos k-mean [3] y los modelos ocultos de Markov (HMM) [3].

2.3. Mantenimiento Predictivo

El entorno competitivo de las industrias, ha obligado a las mismas a entrar en una constante búsqueda de mejores estrategias que sirvan como fuentes de reducción de costos y mejora de performance. Dichas estrategias son orientadas a la minimización de tiempos muertos en la producción, costos de operación, materiales, insumos, repuestos, mantenimientos correctivos y preventivos.

El mantenimiento correctivo es costoso y puede ser riesgoso para el personal que lo realiza. En cambio, el mantenimiento preventivo presenta la ventaja de ser más efectivo y económico que el mantenimiento correctivo. Como desventaja presenta que muchas veces se realiza cambios de piezas en buen estado debido a un mal análisis. Por este motivo se han desarrollado estrategias de mantenimiento más eficientes como el mantenimiento predictivo, que se basa en la hipótesis de que es más eficiente atacar las causas y eliminarlas que trabajar siempre en el efecto.

El objetivo es evaluar la condición de funcionamiento de la máquina eléctrica mientras la misma está en funcionamiento, para evitar detener la máquina y así disminuir los tiempos parados en la producción. Este mantenimiento se basa en pronosticar una falla antes de su ocurrencia, buscando detectar posibles fallas antes de que sucedan utilizando monitoreo de variables y parámetros físicos.

Cuando se realiza el análisis del estado de salud de la máquina o equipo, se estiman distintos síntomas a través de varias metodologías como el análisis de aceites, ultrasonido, termografía y análisis de vibraciones. El análisis de vibraciones sigue siendo el método más utilizado para implementaciones de mantenimiento predictivo.

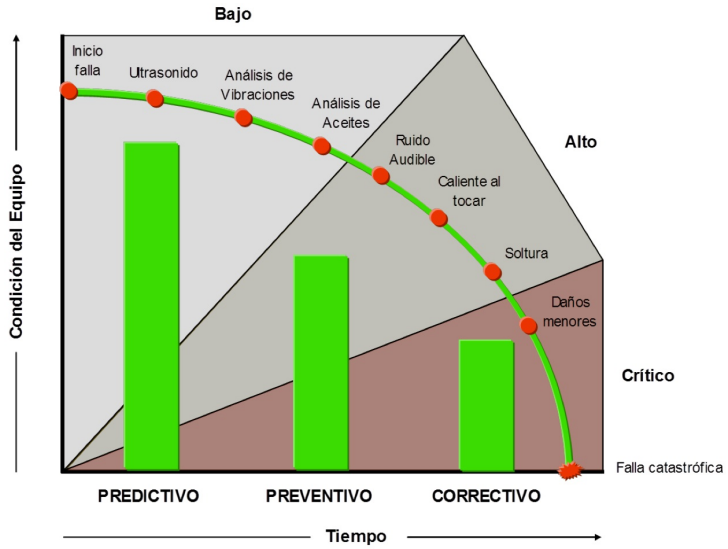


Figura 2.1: Curva de Mantenimiento predictivo

El uso de técnicas de mantenimiento predictivo en grandes industrias del mundo está asociado a la necesidad de cumplir con el objetivo de garantizar que todos los sistemás estén operando dentro de los parámetros establecidos y el mayor tiempo de funcionamiento con el fin de garantizar la máxima productividad de sus equipos. El uso de metodologías de mantenimiento moderno contribuye a maximizar la disponibilidad de los equipos y con ello minimizar los tiempos muertos en la producción.

2.4. Vibraciones

Una vibración es la propagación de ondas elásticas produciendo deformaciones y tensiones sobre un medio continuo. Toda máquina eléctrica posee una señal de vibración donde se obtienen características relacionadas con el movimiento.

Las máquinas rotativas están compuestas por un conjunto de mecanismos diseñados para cumplir con tareas específicas, que van desde una parte que genera el movimiento y otra parte que lleva la energía de los mecanismos que conforman (eje). El eje coordina el funcionamiento, ya que determina la velocidad de giro y es el encargado de transmitir las vibraciones a los mecanismos como los rodamientos, ruedas dentadas, acoples y soportes. La base principal de las señales de vibración para el dominio del tiempo son las ondas senoidales como la Figura 2.2.

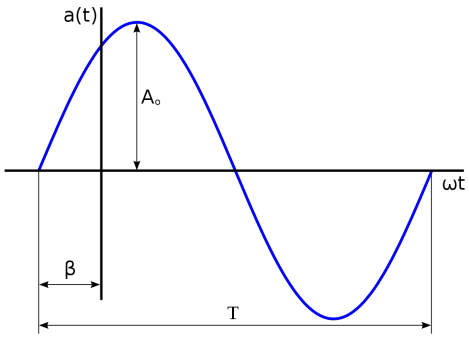


Figura 2.2: Onda de vibración

El nivel de vibración puede estar expresado en término de la aceleración (g), la velocidad (mm/s) o el desplazamiento (μm), mediante los cuales se puede calcular la amplitud de un espectro de la señal de vibración. La variable aplicada a la medida de la vibración permite distinguir valores pico de frecuencias medias y altas, la velocidad permite reconocer valores pico de frecuencias medias y bajas y la de desplazamiento permite distinguir patrones en frecuencias muy bajas.

El uso de vibraciones mecánicas aparece en el campo del mantenimiento en la industria de la mano del mantenimiento preventivo, que surge de la necesidad de la demanda por los elementos vibrantes en una máquina eléctrica y la necesidad de prevenir posibles fallas que traen las mismas. Caso contrario, las vibraciones mecánicas pueden influir en el normal funcionamiento del equipo y provocar una mayor carga en el funcionamiento de la misma, así como desgaste en las piezas del mismo y fatiga la cual puede inducir fallas en las mismas. Dentro de las causas más habituales de falla se puede encontrar la falta de balanceo, excentricidad, falla en rodamientos, problema de engranajes y correas de transmisión.

La presencia de vibraciones en una máquina eléctrica no es un buen indicativo, pero también pueden servir como indicador del estado de salud de la misma, ya que cuando el nivel de vibraciones es bajo, puede indicar que la máquina está operando eficientemente, mientras que si el nivel de vibraciones es elevado puede indicar una alerta de que se aproxima algún tipo de falla/anomalía. Cuando ocurre una falla grave, es probable que se pudiera predecir mediante el cambio gradual de la vibración de la misma.

Como gran cantidad de las fallas de las máquinas vienen predichas por una variación en los niveles de vibración de la misma, la medición y análisis de las vibraciones provee una herramienta fundamental en el mantenimiento predictivo para máquinas eléctricas y equipos en la industria.

2.5. Acelerómetros

El acelerómetro piezoeléctrico (PZT) se puede considerar como el transductor estándar para la medición de vibración en máquinas. Se produce en varias configuraciones, pero el tipo a compresión sirve para describir el principio de operación. Una masa sísmica está sujeta a la base con un perno axial, que se apoya en un resorte circular. El elemento piezo eléctrico está ajustado entre la base y la masa. Cuando está sujeto a una fuerza, se genera una carga eléctrica entre sus superficies como se ve en la Figura 2.3.

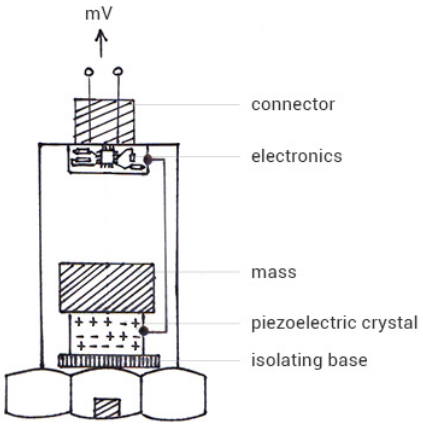


Figura 2.3: Acelerómetro piezoeléctrico

Cuando se mueve el acelerómetro hacia arriba o hacia abajo, la fuerza requerida para mover la masa sísmica está soportada por el elemento activo. Según la segunda ley de Newton, esa fuerza es proporcional a la aceleración de la masa. La fuerza provocada sobre el cristal produce una señal de salida, que es proporcional a la aceleración del transductor. Estos son lineales en el sentido de amplitud, lo que indica que tienen un rango dinámico muy largo. Los niveles de aceleración más bajos están determinados por el ruido electrónico del sistema y los niveles más altos por la destrucción del mismo elemento piezo eléctrico.

El mercado del condition-based monitoring (CbM) ha experimentado un crecimiento significativo en los últimos años. Este crecimiento coincide con el rápido avance de los acelerómetros MEMS para su uso en aplicaciones de detección de vibraciones, que compiten con el acelerómetro piezoeléctrico, dominantes en este tipo de aplicaciones. Existe una mayor demanda de CbM en equipos menos críticos, así como una creciente tasa de adopción de sistemas de CbM inalámbricos, y los acelerómetros MEMS son la clave para ello.

Los sensores de vibración se han utilizado para detectar el estado de máquinas eléctricas desde la década de 1930. Incluso ahora el análisis de vibraciones se considera la modalidad más importante para el mantenimiento predictivo (PdM). Los acelerómetros piezoeléctricos se han establecido desde hace mucho tiempo como el sensor de vibración de referencia utilizado en los equipos más críticos para garantizar que sigan siendo operativos y tengan un rendimiento eficiente. Hasta hace poco, el limitado ancho de banda de los acelerómetros MEMS, su rendimiento y su rango de aceleración (g) impedían su uso en el PdM de equipos críticos. Aunque existen muchos acelerómetros de alto rango de aceleración (diseñados específicamente para la detección de impactos en automóviles), su rendimiento y su ancho de banda son muy limitados, por lo que no son adecuados para la CbM. Asimismo, existen algunos acelerómetros MEMS de bajo ruido (diseñados específicamente para detectar la inclinación), pero su ancho de banda y rango de aceleración son insuficientes.

El ancho de banda de un sensor de vibraciones suele estar relacionado con la criticidad del equipo que va a supervisar. Un equipo o motor crítico es crucial para mantener un proceso o una máquina más grande operativa y en línea. Si un equipo de este tipo se avería, provocaría un tiempo de inactividad no planificado y una posible pérdida de ingresos. Para detectar y diagnosticar los fallos lo antes posible, y evitar tiempos de inactividad imprevistos, es imprescindible contar con un sensor de vibraciones de gran ancho de banda y bajo nivel de ruido.

Se requiere un bajo nivel de ruido para detectar fallos o desviaciones a bajas magnitudes y amplias frecuencias, ya que los fallos habituales en los rodamientos producen a frecuencias más altas, superiores a 5 kHz e incluso hasta 20 kHz y más. Por lo tanto, los sensores MEMS deben ser capaces de competir con los sensores de vibración utilizados de facto durante décadas en las aplicaciones industriales: los acelerómetros piezoeléctricos. Un nivel de ruido de menos de $100ug/\sqrt{Hz}$ y un ancho de banda de 5 kHz o superior se considera un acelerómetro MEMS de alto rendimiento para CbM.

2.6. Redes Neuronales

2.6.1. Neurona

Consideremos un problema de aprendizaje profundo donde tenemos $(x^{(i)}, y^{(i)})$. Las redes neuronales dan una manera de definir una forma compleja, no lineal de hipótesis $h_{W,b}(x)$, con parámetros W, b que podemos ajustar a nuestros datos.

Para describir redes neuronales hay que comenzar a definir una red simple llamada "neurona". En la Figura 2.4 se muestra un diagrama de una red con una sola neurona:

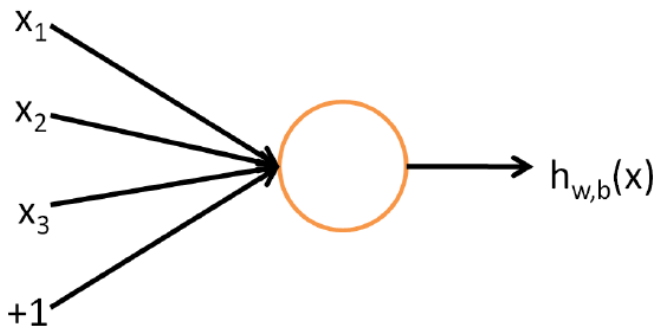


Figura 2.4: Neurona

Esta "neurona" es una unidad computacional que toma como entrada x_1, x_2, x_3 (y $a + 1$ termino de la ecuación de la recta) que produce $h_{W,b}(x) = f(W^T x) = f\sum_{i=1}^{3}(W_i x_i + b)$, donde $f : \mathbb{R} \to \mathbb{R}$ es llamado la **función de activación**. Se elige $f(.)$ para ser la función **sigmoide**:

$$f(z) = \frac{1}{1 + \exp(-z)} \tag{2.1}$$

Por lo tanto, esta neurona corresponde al mapeo de entrada-salida definido por regresión logística. Aunque se utiliza la función sigmoide, cabe aclarar que existen otras funciones para f como lo es la función tangente hiperbólica, o tanh, función:

$$f(z) = \tanh(z) \ = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.2}$$

Se puede observar una comparación de la función sigmoide y la función tanh:

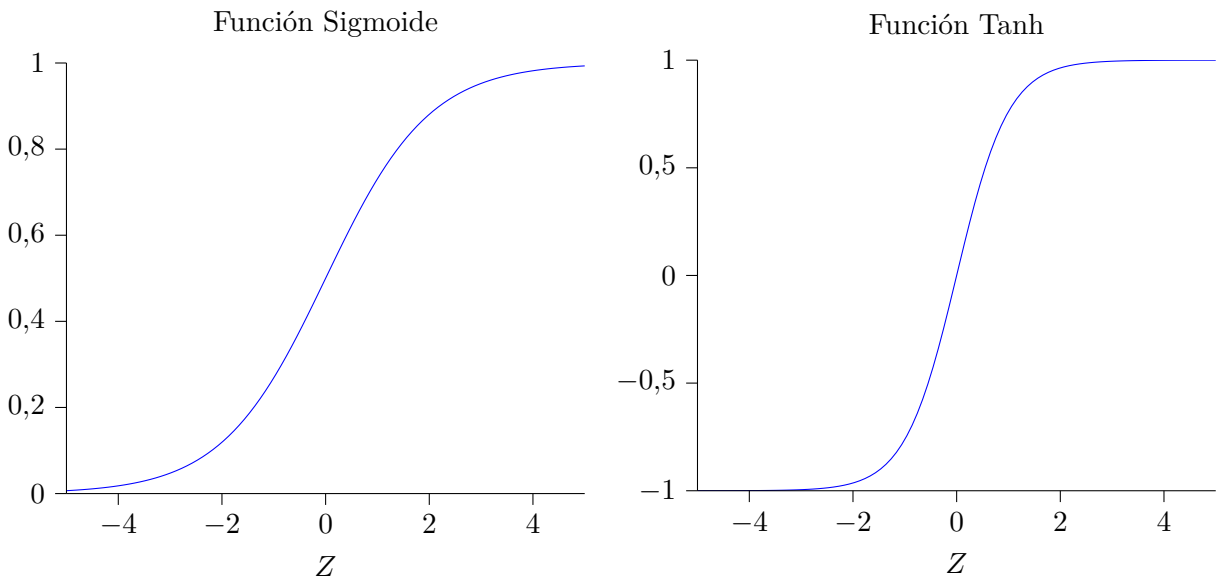


Figura 2.5: Función sigmoide vs Función tanh

La funciónn $tanh(z)$ es una versión escalada de la función sigmoide, y su rango de salida es $[-1, 1]$ en lugar de $[0, 1]$.

2.6.2. Red neuronal

Una red neuronal se arma uniendo a varias de neuronas simples, de modo que la salida de una neurona puede ser la entrada de otra. Se puede observar una red neuronal simple:

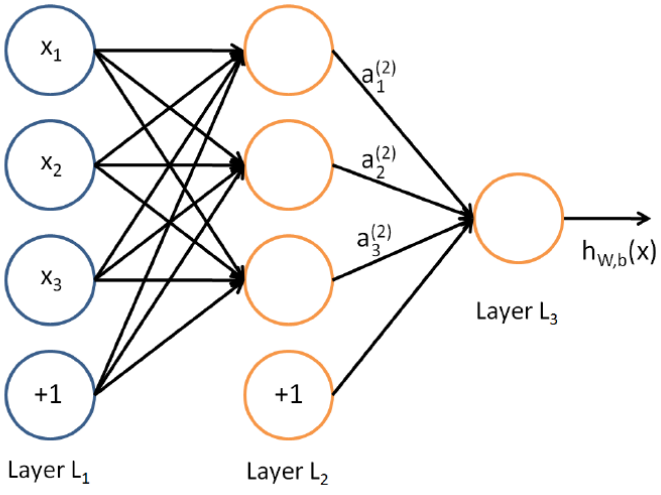


Figura 2.6: Red neuronal simple

En la Figura 2.6, se utilizan circulos para denominar los inputs de la red. Los circulos "+1" se llaman **unidades de bias**, y corresponden al termino de la intercepción. La capa izquierda

de la red se llama la **capa de entrada**, y la capa derecha la **capa de salida** (que, en este caso, tiene solo una neurona). La capa media de nodos se llama la **capa oculta**, porque sus valores no se observan en el conjunto de entrenamiento. En esta representación la red posee 3 entradas (no contando la unidad de bias), 3 nodos ocultos, y 1 unidad de salida. Cabe aclarar que la red puede tener M capas ocultas.

2.6.3. Autoencoders

Una red neuronal autoencoder es un algoritmo de aprendizaje no supervisado que aplica retropropagación, estableciendo los valores de objetivo a ser iguales a los valores de entrada. Es decir, usa $y_i = x_i$.

Es decir, un autoencoder:

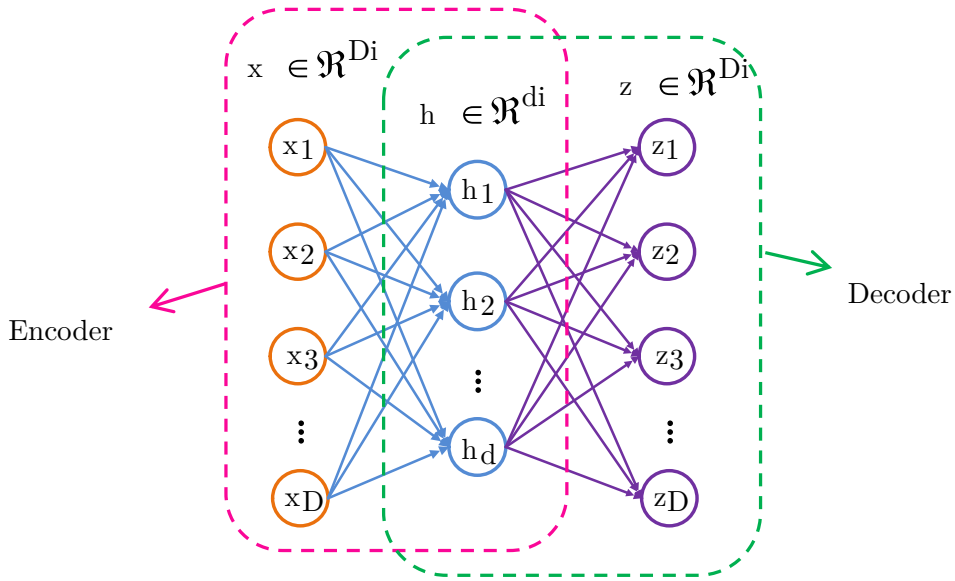


Figura 2.7: Autoencoder

Un autoencoder trata de aprender una función $h_{W,b}(x) \sim x$. Es decir, trata de aprender una aproximación a la función identidad, por lo que se intenta obtener una salida $\hat{x}$ similar a x. La función identidad parece una función bastante trivial para ser tratada; pero al colocar restricciones en la red, como por ejemplo limitar el número de unidades ocultas, se puede descubrir patrones interesantes sobre los datos.

Supongamos que las entradas x son la intensidad de los valores de los píxeles de una imagen de $10x10$ (100 píxeles), por lo que $n = 100$, y hay 50 unidades ocultas en la capa L2. También tenemos $y \epsilon \mathbb{R}100$. Por lo tanto, la red es forzada a aprender una representación comprimida de la entrada. Dado sólo la activación de las unidades ocultas $a^{(2)} \epsilon \mathbb{R}50$, debe intentar **reconstruir** la entrada de 100 píxeles. Si la entrada fuera completamente aleatoria, es decir, cada x_i viene de una distribución independiente de las otras características, entonces esta tarea de compresión sería muy difícil. Pero si hay estructura en los datos, por ejemplo, si algunas de las características de entrada están correlacionadas, entonces este algoritmo puede descubrir algunas de esas correlaciones.

El argumento planteado anteriormente se refiere a que el número de unidades ocultas sea

pequeño. Pero incluso cuando el numero de unidades ocultas es grande (incluso mayor que el número de píxeles de entrada), se puede descubrir una estructura interesante, aplicando otras restricciones en la red. En particular, si se establece una restricción de esparcimiento (**sparsity**) en las unidades ocultas, entonces el autoencoder sigue descubriendo una estructura interesante en los datos, incluso si el número de unidades ocultas es grande.

Se piensa que una neurona es "activa" si su valor de salida está cerca de 1, o es "inactiva" si su valor de salida está cerca de 0. Se busca que la mayoría de las neuronas esten inactivas.

Recordando que $a_j^{(2)}$ denota la activación de la neurona oculta j en la red autoencoder. Esta notación no hace explicito la entrada x que llevó a esa activación, por lo tanto, se escribe $a_j(x)$ para denotar la activación de esta neurona oculta cuando la red es posee una entrada específica x. Además:

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[a_j^{(2)}(x^{(i)}) \right] \tag{2.3}$$

es la activación media de la neurona oculta j (promediada sobre el conjunto de entrenamiento). Se busca forzar la restricción $\hat{\rho}_j = \rho$, donde ρ es un parámetro de esparcimiento, que generalmente es un valor cercano a cero (ejemplo $\rho = 0{,}05$). Por lo tanto, se busca que la activación media de cada neurona oculta j esté cerca de 0.05. Para lograr esto, se añade una penalización extra al objetivo de optimización que penaliza a los $\hat{\rho}_j$ que se desvian significativamente de ρ. Existen varias formas de penalizar este comportamiento que ofrecen buenos resultados, siendo uno de ellos:

$$\sum_{j=1}^{s2} \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j} \tag{2.4}$$

En la ecuación 2.4, s_2 es el numero de neuronas en la capa oculta y el indice j va sumando en base a las unidades ocultas de la red. Este termino, se basa en la divergencia KL, la cual puede expresarse de la siguiente manera:

$$\sum_{j=1}^{s2} KL(\rho || \hat{\rho}_j) \tag{2.5}$$

donde $KL(\rho||\hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j}$ es la divergencia Kullback-Liebler (KL) entre una variable aleatoria de Bernoulli con media ρ y una variable aleatoria de Bernoulli de media $\hat{\rho}_j$. La divergencia KL es una función estándar para medir cuan diferente dos distribuciones son.

Esta función de penalidad tiene la propiedad de que $KL(\rho||\hat{\rho}_j) = 0$ si $\hat{\rho}_j = \rho$, y aumenta monótonamente si $\hat{\rho}_j$ diverge de ρ. Por ejemplo, en la siguiente figura, se establece $\rho = 0{,}2$, y se grafica $KL(\rho||\hat{\rho}_j)$ para un rango de valores de $\hat{\rho}_j$:

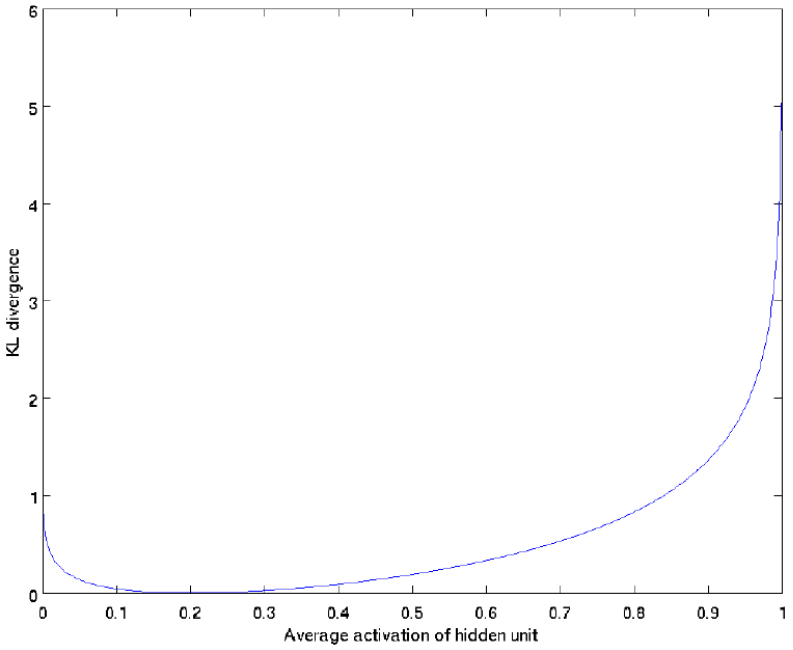


Figura 2.8: Divergencia KL

Se observa en la Figura 2.8 que la divergencia KL alcanza un minimo de 0 en $\hat{\rho}_j = \rho$ y crece exponencialmente cuando $\hat{\rho}_j$ se acerca a 0 o 1. Por lo tanto, minimizar este termino tiene el efecto de causar que $\hat{\rho}_j$ sea cercano a ρ.

La función de costo de un autoencoder utilizando la divergencia KL es la siguiente:

$$J_{sparse}(W,b) = J(W,b) + \beta \sum_{j=1}^{s2} KL(\rho||\hat{\rho}_j) \tag{2.6}$$

donde $J(W,b)$ fue definido previamente y β controla el peso del sparsity del Autoencoder. Luego el termino $\hat{\rho}_j$ (implicitamente) depende de (W,b).

2.7. Aceleradores de Hardware

2.7.1. Aprendizaje profundo

El aprendizaje profundo (deep learning) es un tipo de machine learning que entrena a una computadora para realizar tareas de manera más similar a como hacemos los seres humanos, como el reconocimiento de voz, identificación/clasificación de imágenes o hacer predicciones basándose en datos. En el aprendizaje profundo, se configuran varios parámetros básicos acerca de los datos de entrada y entrena la red neuronal para que aprenda por su cuenta por lo que luego es capaz de ir reconociendo patrones mediante el uso de distintas capas de procesamiento.

Se necesita mucho poder de procesamiento para poder resolver problemas implementados con aprendizaje profundo debido a que los algoritmos de aprendizaje profundo son de manera iterativa. Su complejidad aumenta con el número de capas y con los datos de entrada que se necesitan para entrenar a las redes.

El algoritmo de entrenamiento es un mecanismo que, a partir de un conjunto de patrones, encuentra los parámetros de la red para que responda de manera satisfactoria cuando se le presentan dichos patrones (set de entrenamiento). Los parámetros son los "pesos sinápticos", es en estos parámetros donde queda el conocimiento de la red.

Las redes neuronales artificiales son buenas para tareas perceptuales y asociaciones. La red puede generalizar y así tratar con ejemplos no conocidos, son tolerantes al ruido por lo que pequeños cambios en la entrada no afectan drásticamente la salida de la red.

El flujo de información y aprendizaje de la red se basa en el algoritmo de *backpropagation* [7] el cual es un método de cálculo del gradiente utilizado en los algoritmos de aprendizaje de redes neuronales. El método emplea un ciclo de propagación-adaptación de dos fases. Una vez que se aplicó un patrón a la entrada de la red como estímulo, se propaga desde la primera capa a través de las demás hasta llegar a la salida. La señal de salida se compara con la señal de salida deseada y se calcula una señal de error para cada una de las salidas.

La importancia de este proceso radica en que mientras se entrena la red, las neuronas de capas intermedias se adaptan a sí mismas de tal manera que distintas neuronas aprenden a reconocer distintas características del espacio total de entrada. Una vez finalizado el entrenamiento, cuando se presente un patrón de entrada que contenga ruido o esté incompleto, las neuronas presentes en la capa oculta se activarán si la nueva entrada posee un patrón que se asemeje a la característica aprendida durante su entrenamiento.

2.7.2. Arquitectura de las FPGA

La arquitectura de la FPGA son arreglos de bloques lógicos programables además de elementos de memorias conectados entre sí mediante interconexiones programables. Estos bloques se implementan como "tablas de consulta" (LUTs: Look Up Tables), que son memorias cuya señal son las entradas y las salidas están pre calculadas y almacenadas en la LUT. Una LUT puede programarse para calcular funciones booleanas al usar una tabla de verdad como los valores de la memoria LUT, y por lo tanto almacenar resultados de una función representable en bits. Esto permite acelerar considerablemente muchos cálculos dado que es más eficiente buscar un resultado en una tabla que calcularlo [2].

Los bloques Flip-Flop (FF) son los elementos de memoria básica de las FPGA, y generalmente están junto a las LUTs. Cuando existen varias LUTs y varios FF conectados (junto a otros elementos como sumador o multiplicador), se obtiene un elemento lógico más complejo llamado bloque lógico configurable. En la arquitectura Xilinx, el nombre usado es slice, donde un slice es la combinación de LUTs, FF y multiplexores combinados. El número exacto de cada elemento depende del modelo y la arquitectura, pero en general son pocos componentes [2]. Se puede observar en la siguiente imagen el funcionamiento de los slices:

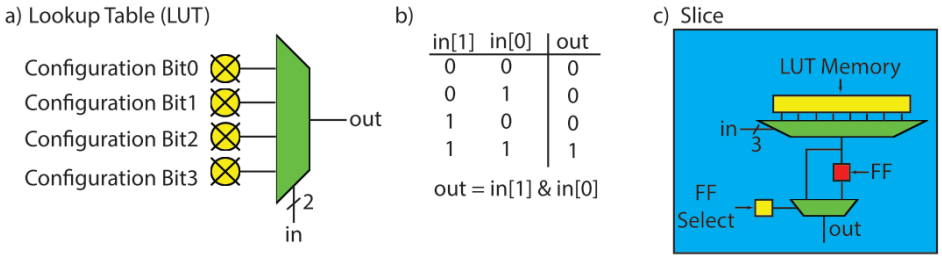


Figura 2.9: Diagrama de LUT y Slice

Las interconexiones programables son otro elemento de las FPGA, ya que permiten una red flexible para crear conexiones entre slices. Los inputs/outputs de cada slice están conectados a un canal de enrutamiento, que contiene una serie de bits de configuración para conectar o desconectar el bloque slice a la interconexión programable. Los canales están conectados a switchboxes, que consisten en transistores que permiten conectar los canales entre sí. La siguiente imagen presenta un ejemplo de interconexión de los componentes:

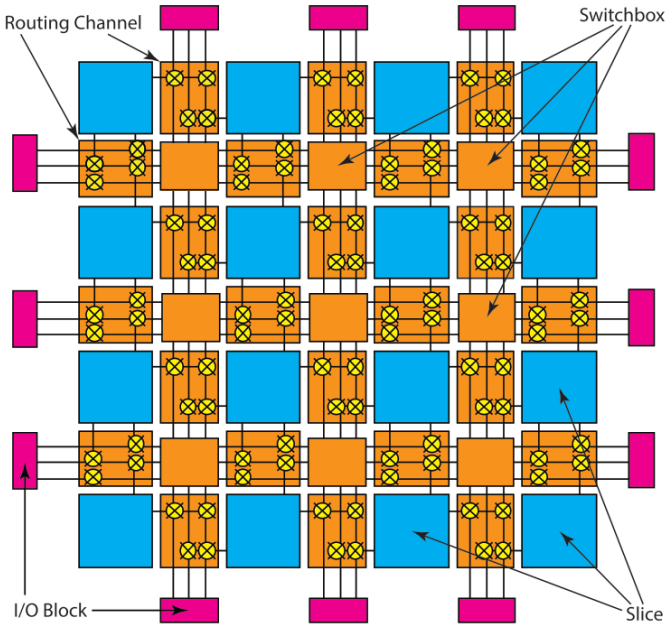


Figura 2.10: Interconexión de bloques

Los bloques I/O son interfaces externas, como lo son un procesador, memoria, sensor o actuador. No es necesario comprender el detalle de funcionamiento de cada componente de una FPGA si se utiliza síntesis de alto nivel (HLS), pero si conviene poseer una comprensión general de los recursos para saber el impacto de las directivas de optimización.

Cada vez las FPGA van incorporando más elementos en su arquitectura como bloques RAM (BRAM) o DSPs. Un bloque BRAM es una RAM configurable que puede implementar distintas configuraciones de memoria e interfaces, como almacenar bytes, half words, words, double words y conectarse a buses on-chip y buses del procesador. Dependiendo la cantidad de datos necesarios

para almacenar en la FPGA se pueden elegir distintos modelos de placa.

	Memoria Externa	**BRAM**	**FFs**
Cantidad	1-4	Miles	Millones
Tamaño	GBytes	KBytes	Bits
Tamaño Total	GBytes	MBytes	100s de KBytes
Ancho	8-64	1-16	1
Total Ancho de Banda	GBytes	TBytes/s	100s de TBytes

Tabla 2.1: Recursos de FPGA [2]

Podemos observar en la siguiente tabla una comparación entre distintos modelos de FPGA:

	Logic Slices	**BRAM**	**DSP Slices**	**FFs**
Zybo Z7-10	4,400	270Kb	80	35,200
Pynq Z2	13,300	630Kb	220	106,400
ZCU102	599,550	32.1Mb	2,520	548,160

Tabla 2.2: Comparación de recusos en modelos de FPGA

Recordemos que la placa Zybo Z7-10 es la utilizada para este proyecto, siendo la de menor recursos disponibles en la Tabla 2.2 lo cual presentará algunas limitaciones que serán analizadas en los siguientes capítulos. Sin embargo, la ventaja de las placas de bajos recursos es el coste, permitiendo una mayor accesibilidad a hardware dedicado así como la capacidad de desarrollar prototipos de bajo costo para evaluar soluciones sin necesidad de requerir un gran presupuesto inicial.

2.7.3. Aprendizaje profundo en FPGA

En contextos experimentales, los sensores han aumentado su precisión y frecuencia de trabajo, permitiendo que cada vez haya más volumen de datos. La mayoría de las veces también deben ser procesados a altas velocidades: filtrado, clasificación y procesamiento en "tiempo real". Una opción para esto es el uso de ASIC, pero también se utilizan mucho las FPGA que son una buena alternativa debido a la flexibilidad y baja latencia que estas placas pueden alcanzar. En algunas aplicaciones, como la detección de partículas en el Large Hadron Collider (LHC) no es posible utilizar GPU o CPU para su procesamiento debido a sus exigencias de tiempo de procesamiento de 25ns (40MHZ) [8].

En estos contextos, es necesario una metodología de implementación de modelos entre modelos de machine learning y su aplicación en FPGA usando herramientas de síntesis de alto nivel. Para esto, existen varios caminos para realizar la arquitectura de la red neuronal y la implementación en FPGA. Por un lado, se puede realizar una programación de bajo nivel en el lenguaje C/C++ y realizar un proceso de síntesis de alto nivel (HLS - High-Level Synthesis), para personalizar las optimizaciones para la arquitectura elegida.

Algunas cosas para tener en cuenta en la implementación de NNs en FPGA son:

- Latencia: es el tiempo total que se demora una iteración del algoritmo.
- Intervalo de iniciación: es el número de ciclos de reloj que se requieren antes de que el algoritmo pueda aceptar una nueva entrada. Es inversamente proporcional a la tasa de

inferencia (o rendimiento máximo).

- Uso de recursos: Cantidad de dispositivos usados: memoria en placa (BRAM), DSPs, registros, flip-flops, LUTs [2].

La implementación de redes neuronales en FPGA es una de las más complejas, ya que se requiere una arquitectura de red neuronal que pueda ser personalizada para cada caso de uso. En este trabajo se realizará una implementación a través de una herramienta de código abierto que analizaremos luego.

Teniendo en cuenta lo nombrado, y también debido a la existencia de diferentes escenarios para aplicaciones de Deep Learning y Redes Neuronales, se evidencia la necesidad de realizar un mapeo rápido entre las redes de DL y NN en FPGA. Con distintas herramientas es posible realizar automatizaciones que realicen este trabajo, teniendo en cuenta parámetros, topología, configuración de la red y las distintas características de la FPGA elegida. Es decir, se busca generar automáticamente la implementación en hardware de la red elegida sin necesidad de estar realizando la programación del hardware manualmente. Esto permite una mayor integración de sistemas de machine learning en ecosistemas con FPGA.

Por esto, existen diferentes herramientas con diferentes características de implementación en cada una, así como ciertos target de placas/modelos de FPGA. Las herramientas soportan capas convolucionales, activaciones no lineales, capas de muestreo y capas fully connected. Algunas herramientas también soportan RNN (Recurrent Neural Network) y LSTM (Long short-term memory), Las capas convolucionales y fully connected constituyen el 99 % del total de las operaciones y pesos de la red [9].

Para realizar la implementación, las herramientas realizan una parametrización de la FPGA objetivo. Dicha herramienta define un espacio de diseño de la arquitectura, en donde un punto en este espacio corresponde a la latencia, rendimiento, uso de recursos y eficiencia energética. Cada herramienta crea un modelo matemático con estas relaciones y busca minimizar mediante diversas optimizaciones y algoritmos alguna de estas variables. Las herramientas permiten configurar o parametrizar estas optimizaciones y lograr una exploración más detallada de los parámetros de la arquitectura.

Estas herramientas además llevan a cabo diferentes optimizaciones en el proceso de inferencia, entre las más comunes se encuentra la reducción de precisión en la representación numérica de las variables (cuantización), donde se pasa de una base de punto flotante de 32 bits a punto fijo de 16 u 8 bis siendo los más comunes. Existen lineales, datos no lineales, datos uniformes y no uniformes [9].

Otra optimización que se suele realizar es la reducción de los pesos de la red, mediante aproximación de matrices de menor dimensión, que se puede realizar por medio de técnicas de análisis factorial y también con la descomposición de valores singulares. También se utiliza la eliminación de pesos debajo de cierto valor umbral (pruning) [9].

2.7.4. Herramienta - HLS4ML

Para este proyecto se eligió la herramienta **hls4ml**. La misma es de código abierto y esto permite un fácil acceso a la misma. Esta herramienta también ofrece flexibilidad a la hora de realizar la optimización, por el lado de la latencia o utilización de recursos, ya que esto permite que puedan coexistir distintos algoritmos para su implementación en FPGA. Por esto, si bien existen varias herramientas para realizar la implementación de redes neuronales en FPGA, se eligió esta herramienta por su flexibilidad en la optimización junto con un rendimiento aceptable.

Y por último, es una herramienta desarrollada en el contexto del Gran Colisionador de Hadrones (LHC), el proyecto de física experimental de mayor envergadura en el mundo actualmente [10].

El proceso de trabajo de la herramienta se puede observar en la siguiente imagen:

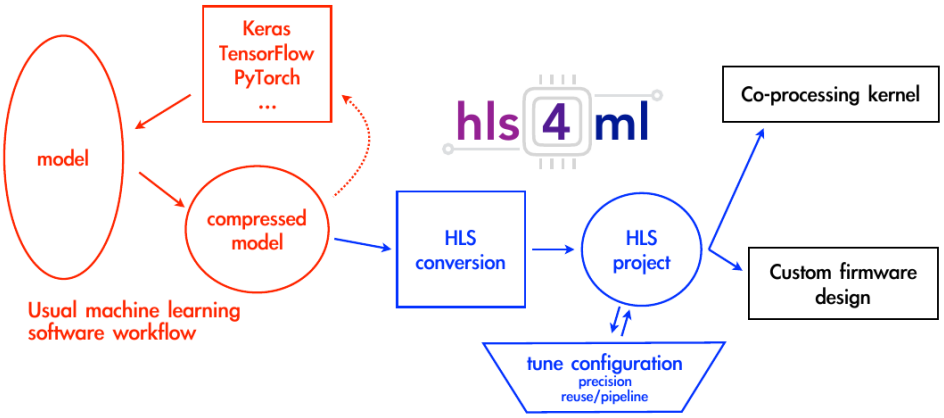


Figura 2.11: Flujo de la herramienta hls4ml

El sistema operativo utilizado para estas herramientas es PYNQ [11], la cual es una plataforma de código abierto de Xilinx que permite una interfaz de trabajo con Python utilizando las APIs existentes para plataformas Xilinx. Los circuitos de lógica programable (PL) son presentados como librerías llamadas overlays. Estos overlays son análogos a librerías de software. Un desarrollador de software puede seleccionar el overlay que mejor se adapte a su aplicación. El overlay puede ser accedido a través de una API de Python.

El único punto en contra de esta herramienta es que la disponibilidad de modelos de FPGA compatibles es muy limitado y el modelo elegido para este proyecto no era compatible. Para ello fue necesario adaptar la herramienta para poder utilizarse con la placa elegida. Fue necesario volver a compilar el sistema operativo base, reconfigurar las directivas de compilación propias para la placa de trabajo Zybo Z7-10 y luego realizar la compilación final del sistema PYNQ para que pudiera soportar la configuración de este modelo de FPGA. Si bien parte de la funcionalidad que ofrece PYNQ no quedó compatible con la arquitectura de la nueva placa, se pudo adaptar las partes necesarias para el funcionamiento del proyecto.

2.7.5. Flujo de trabajo para la programación de FPGA

En este punto es importante aclarar que existen diversas maneras de programar las FPGA. En esta sección se busca aclarar el flujo de trabajo al programar FPGA utilizando herramientas de síntesis de alto nivel (HLS: High-level Synthesis).

Lo primero es comprender el objetivo final es crear una aplicación de software que contiene ciertas (o todas) las funciones aceleradas mediante el uso de lógica programable (PL Programable Logic) de la FPGA. Esta aplicación puede tener múltiples funciones para obtención de datos, también realizar pre y post procesamiento, uso de pantalla o archivos. La función acelerada por hardware es solo una parte de la aplicación host y comúnmente recibe datos de entrada, los procesa y luego devuelve el resultado, encargándose de solo una parte de la aplicación general.

Normalmente no es necesario acelerar por hardware todos los componentes o funciones de la aplicación, sino identificar aquellas partes que son de uso más intensivo y paralelizable que

se beneficiarían de su aceleración con la PL. La aplicación luego llama mediante el uso de APIs o protocolos a la función acelerada. El siguiente esquema muestra el flujo de trabajo indicado:

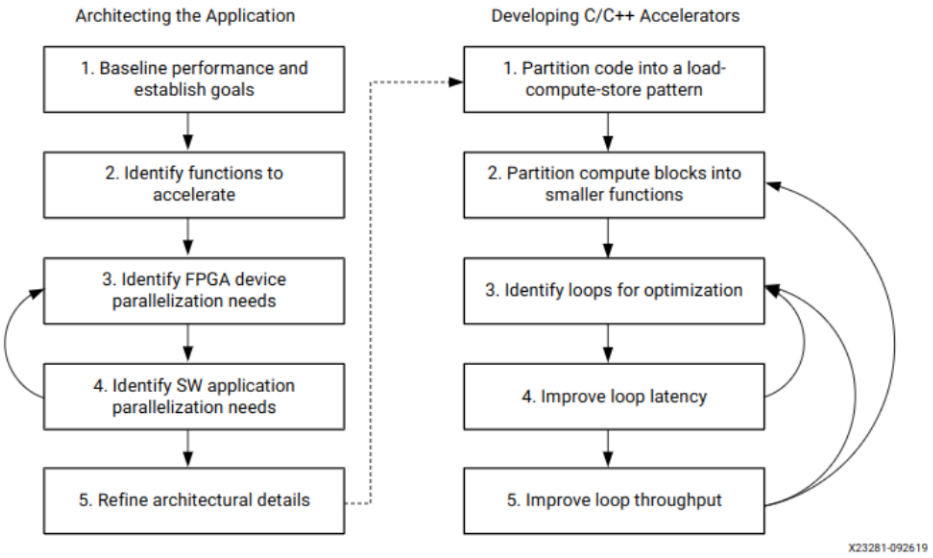


Figura 2.12: Flujo de trabajo para la programación de FPGA [1]

Estos procesos corresponden a herramientas y flujos distintos. Una parte se basa en el desarrollo de la aplicación host para FPGA de Xilinx que se puede realizar con Xilinx SDK o Vitis y consiste en una aplicación en lenguaje C/C++ embebido. Esta aplicación consta de una función principal y además deben realizarse la carga de librerías, inicialización de controladores de dispositivos de hardware creados para la aceleración, carga de datos, procesamiento, pero también es posible que se realicen funciones de más alto nivel como impresión en pantalla de resultados, cálculos de tiempo, entre otros.

Por otro lado, la función acelerada por hardware se puede implementar utilizando la HLS (síntesis de alto nivel). Esta función acelerada consiste en una función que tiene parámetros, escalares o de vectores de entrada, y escalares o de vectores de salida. Esta función es la que se busca implementar en el PL, ya que se busca paralelizarla y acelerarla. Dicha función está escrita en C/C++ y su ejecución se optimiza mediante distintas directivas que se agregan al código. Este es el centro del presente trabajo, ya que es la funcionalidad que permite correr la red neuronal (para este proyecto) en la FPGA objetivo.

2.7.6. Síntesis de Alto Nivel (HLS)

El proceso de diseño de hardware (HW) fue evolucionando a lo largo de la historia. Al comienzo, cuando los circuitos eran pequeños, era posible especificar cada elemento, conexiones y todo era de manera manual. A medida que fue creciendo la complejidad de los circuitos, fueron apareciendo herramientas automatizadas de modo que cada vez fue posible lograr niveles más altos de abstracción, especificando circuitos en vez de componentes individuales.

En los años 80 apareció la aproximación de Mead y Conway, que comenzó la descripción de hardware como Verliog o VHDL que se usan para describir sistemas digitales. A medida que la complejidad siguió aumentando, apareció el nivel RTL (Register-Transfer Level) que permitía un nivel más alto de abstracción, permitiendo especificar registros y operaciones que

se realizan en estos registros. La automatización permitía traducir estas especificaciones en circuitos complejos.

La síntesis de alto nivel (HLS) permite un nivel más de abstracción, ya que permite trabajar en partes más generales de la arquitectura en vez de estar enfocado en registros u operaciones. Esto permite trabajar sobre el comportamiento de un programa que no especifica registros o ciclos y HLS permite trabajar de manera automática las siguientes tareas:

- Analizar y aprovechar la concurrencia de algoritmos.
- Insertar registros según sea necesario en caminos críticos.
- Generar una lógica de control que controla el flujo de datos.
- Implementar interfaces de interconexión del sistema.
- Mapeo de datos en elementos de almacenamiento.
- Mapeo de elementos lógicos para llevar a cabo optimizaciones automáticas o especificas del usuario.

Como se había comentado, el programa o algoritmo es escrito en un lenguaje de alto nivel como C/C++. Luego, se configura la micro-arquitectura a través del agregado de directivas de interfaz y optimización en ciertos segmentos del código para indicar al programa donde debe realizar las optimizaciones. La herramienta utilizada, Vivado HLS permite (entre otras) las siguientes especificaciones:

- Código o funciones en C/C++ que se desean acelerar.
- Ejecución de tests sobre el código a través de Test Bench para verificar la correcta implementación.
- La placa/modelo FPGA que se desea utilizar.
- La velocidad de trabajo elegida.
- Uso de directivas para optimizar el código.

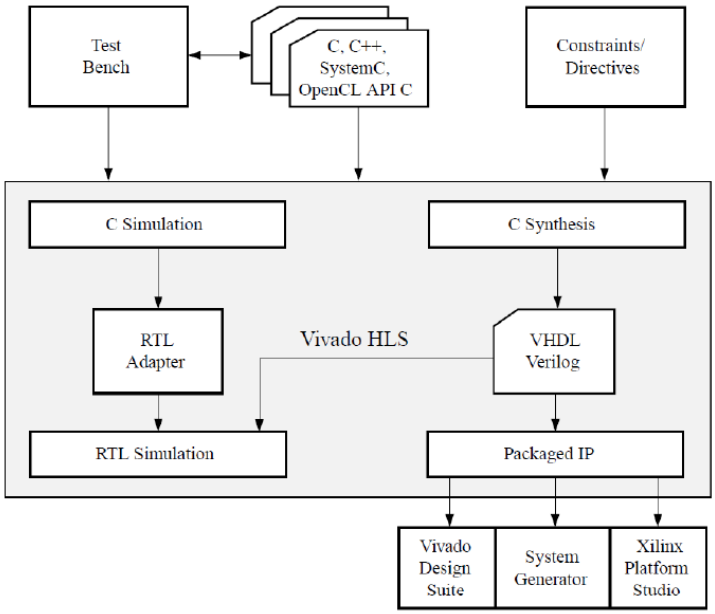


Figura 2.13: Diagrama HLS [1]

En la Figura 2.13 podemos observar el diagrama de funcionamiento de HLS. Para el flujo de diseño se comienza con la función de C/C++ que se desea acelerar, es necesario diseñar el código cuidadosamente y aplicar optimizaciones, el objetivo es lograr una función acelerable en lenguaje de bajo nivel (VHDL o Verilog).

Tomemos de ejemplo un caso base de diseño con HLS como lo es sumar dos enteros de 32 bits. El código de la función sumar es el siguiente:

```
void add(int a, int b, int& c) {
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE s_axilite port=a
    #pragma HLS INTERFACE s_axilite port=b
    #pragma HLS INTERFACE s_axilite port=c

    c = a + b;
}
```

La función responde a una función clásica del lenguaje C/C++ con la particularidad de que se declara la *interfaz* de los puertos a utilizar para el input/output de los datos además de una línea para desactivar protocolos de control de I/O que no son necesarios para este caso. Una vez compilado y sintetizado, se genera un bloque IP:

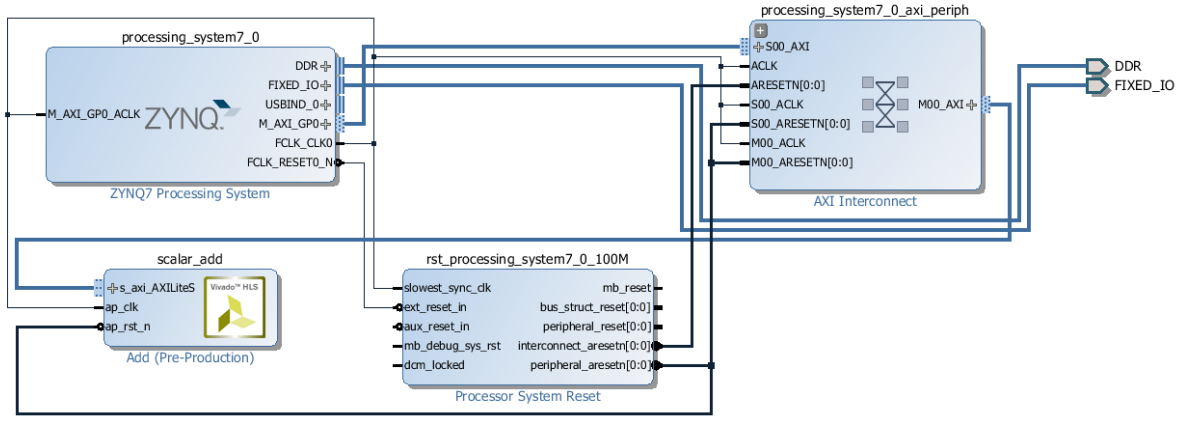


Figura 2.14: Diagrama en bloque HLS

En la Figura 2.14 se puede observar el bloque IP generado mediante HLS *scalar_add* que luego se integra mediante el software de Vivado a otros bloques necesarios para poder ejecutar la función diseñada en la placa como lo es el procesador ZYNQ7. Para poder interactuar con el código es necesario programar la placa. Para ello se sintetizan los bloques en hardware en un archivo bitstream que contiene las intrucciones de programación de la FPGA. Este archivo se escribe y ejecuta en la plata y luego se utiliza un programa en C/C++ para controlar el funcionamiento y flujo de datos de la FPGA. En este proyecto se utiliza el framework de trabajo PYNQ para realizar el programa de control de los bloques IP.

```
1  from pynq import Overlay
2
3  overlay = Overlay('/home/xilinx/scalar_add.bit')
4  add_ip = overlay.scalar_add
5
6  add_ip.write(0x10, 4)
7  add_ip.write(0x18, 5)
8  add_ip.read(0x20)
9
10 # Out: 9
```

En el ejemplo del código, se puede observar como desde un lenguaje de alto nivel como es Python, accedemos en runtime al bitfile generado con anterioridad mediante HLS y luego escribimos en el bloque IP los dos datos que se desean sumar. Luego, se lee el resultado de la suma en el bloque IP y se imprime en pantalla. Este es el método por el cual se realizará la interfaz de la red neuronal para mayor simpleza a la hora del control de los datos necesarios para su procesamiento.

3 Autoencoder para detección de fallas

3.1. Adaptación del método

Ya se explicó el funcionamiento de un Autoencoder en la sección 2.6.3. Es necesario diseñar una red para la aplicación en este proyecto. Tomando de referencia la investigación original [12] se puede observar que el autoencoder tradicional no posee buena respuesta en cuanto al ruido de entrada que puede afectar el funcionamiento de la red. Para ello es necesario implementar una nueva función de costo para la red neuronal a utilizar para evitar este problema.

La función de perdida de un Autoencoder estándar es la función de costo MSE (Mean Squared Error) la cual no es robusta para aprender características de señales complejas [13]. La **Correntropía** es una medida no lineal y local de similitud, y la **Máxima Correntropía** es inmune para ruido de fondo complejo y no estacionario [14, 15]. Por lo tanto, la **Máxima Correntropía** tiene el potencial de hacer coincidir las características de una señal compleja, lo que puede resolver los problemas de MSE.

Para dos variables $A = [a_1, a_2, ...a_N]^T$ y $B = [b_1, b_2, ...b_N]^T$, la **correntropía** se define como:

$$V_\sigma(A, B) = E[\kappa_\sigma(A, B)] = \int \kappa_\sigma(a, b) dF_{AB}(a, b) \tag{3.1}$$

El Kernel Gausiano más popular para correntropía es el Kernel Mercer, el cual se define como:

$$\kappa_\sigma(a_i, b_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(a_i - b_i)^2}{2\sigma^2}\right) \tag{3.2}$$

donde σ es el tamaño del kernel. Luego, la nueva función del Autoencoder se puede diseñar maximizando la función:

$$J_{MC}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \kappa_\sigma(\mathbf{X}_i - \mathbf{Z}_i) \tag{3.3}$$

Por último, se agrega un termino $J_{weight}(\theta)$ para regularizar los pesos de la red y evitar *overfitting*, expresado como:

$$J_{weight}(\theta) = \frac{\lambda}{2} \sum_{l=1}^{2} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} W_{ij}^2 \tag{3.4}$$

siendo W_{ij}^l un elemento en $\mathbf{W}^l$ y λ un parámetro que ajusta el peso de J_{weight}, s_l el número de unidades en la capa l.

Similar al Autoencoder estándar, la nueva función del Autoencoder trata de minimizar el error de reconstrucción optimizando sus parametros. Considerando el criterio de Maxima Correntropía (ecuación 3.3) trata de maximizar la correntropía entre una variable y su estimador, en orden de minimizar el error de reconstrucción y maximizar la correntropía al mismo tiempo. La nueva función de costo para este proyecto queda definida como:

$$J_{new}(\theta) = -J_{MC}(\theta) + J_{weight}(\theta) + J_{sparse}(\theta) \tag{3.5}$$

Donde veremos la implementación de cada término de la ecuación 3.5 en las siguientes secciones de la tesis, así como su implementación.

3.2. Caso de estudio

Se plantearon los beneficios que posee la red de Autoencoder y el software para realizar la implementación. A fin de comenzar con las pruebas es necesario primero validar y verificar una red neuronal que se adapte al problema planteado para analizar y clasificar vibraciones de un motor.

Para ello se comenzó con un dataset llamado Case Western Reserve University [16] (CWRU). La información fue obtenida de una serie de acelerómetros acoplados a un sistema mecánico de un motor de inducción de 2HP 3.1 en conjunto con un transductor de torque en el medio y un dinamómetro conectado a la derecha. Se instalaron dos acelerómetros sampleando a una frecuencia de 12 kHz y 48 kHz. En este dataset se introdujeron fallas de manera artificial en los rodamientos, teniendo cuatro tipos de fallas: falla de bola, falla de rueda interna, falla de rueda externa y falla de rueda. Cada falla tiene un diámetro de 0.007, 0.014 y 0.021 pulgadas respectivamente, haciendo un total de 10 categorías de condición del motor.

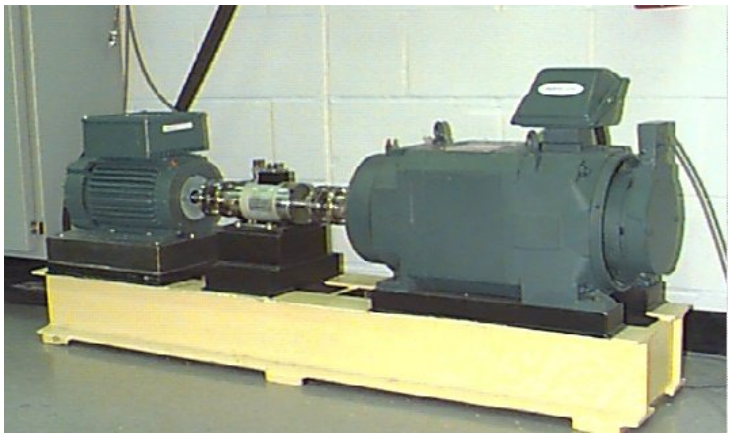


Figura 3.1: Banco de ensayos utilizado en el experimento de CWRU

El dataset de CWRU sirve como un dataset fundamental para validar la performance de distintos algoritmos de machine learning y deep learning [17].

Dado que solamente tener un dataset no resuelve el problema es necesario determinar la topología y características del Autoencoder a utilizar para adaptarlo al problema elegido. Se realizó una investigación de distintos modelos de machine learning entre los muchos que se han investigado para analizar fallas de máquinas eléctricas [17, 18] y se tomó como referencia base la investigación realizada en [12].

En la misma se propone un modelo de Autoencoder con solo la etapa de encoding pero sin la etapa de decoding de la red, cambiando además la función de activación de la red para obtener un clasificador de fallas. Luego de propuesto el método se procede a realizar una validación experimental con un set de datos con un banco de ensayos propio.

Para este proyecto, se realizó una primera validación experimental utilizando el dataset de CWRU y luego se realizó una adaptación de la red para poder ser corrida en la FPGA elegida con

base en los criterios de recursos que requiere una placa FPGA de bajos recursos comparada con los mayores recursos que posee una PC de escritorio para correr programas de redes neuronales.

El proceso de verificación consta en realizar el entrenamiento de la red y luego verificar los resultados con un pequeño set de validación de datos. Una vez entrenado el modelo reducido para la FPGA se corre el programa en la misma de manera acelerada para verificar su funcionamiento. Para este proyecto el alcance se limita a verificar que es posible la implementación de una red neuronal para clasificar fallas de motor en una FPGA, pero no se realiza un software de gestión de fallas.

3.3. Validación Inicial

De la gran cantidad de datos provista por CWRU se tomó un set reducido de datos. El segmento elegido se obtuvo con una carga de 1HP en el eje y las fallas evaluadas son:

- C1 : Ball defect (0.007 inch, carga: 1 hp)
- C2 : Ball defect (0.014 inch, carga: 1 hp)
- C3 : Ball defect (0.021 inch, carga: 1 hp)
- C4 : Inner race fault (0.007 inch, carga: 1 hp)
- C5 : Inner race fault (0.014 inch, carga: 1 hp)
- C6 : Inner race fault (0.021 inch, carga: 1 hp)
- C7 : Normal (carga: 1 hp)
- C8 : Outer race fault (0.007 inch, carga: 1 hp, 6 O'clock)
- C9 : Outer race fault (0.014 inch, carga: 1 hp, 6 O'clock)
- C10 : Outer race fault (0.021 inch, carga: 1 hp, 6 O'clock)

Como se nombró, existen diversos tipos de topologías para implementar métodos de ML y DL. El caso de estudio de [12] propone un método teórico para la implementación de un Autoencoder y luego realiza una validación con datos experimentales. A fin de utilizar un dataset conocido como CWRU se realizó una adaptación de la red propuesta para trabajar sobre este dataset. Luego se realizó la validación experimental.

El largo de un único sample que es provisto como entrada de un autoencoder es denominado tamaño de entrada. Se observa que la calidad de representaciones y características (que son extraídas de la entrada) mejoran cuando se provee una entrada de mayor tamaño al autoencoder [19]. Sin embargo, también aumenta el costo computacional y por lo tanto se utiliza un tamaño de entrada de 1024 para lograr una mejora en la calidad de la representación. El uso de una entrada de mayor tamaño para el Autoencoder mejoraría significativamente el tiempo de ejecución de la red, pero podría no proveer mejoras proporcionales en su diagnóstico.

El número oculto de neuronas que aparecen en la capa oculta de un Autoencoder es un factor clave en la extracción de características representativas de nivel superior. No hay una regla definida para seleccionar el número de nodos en la capa oculta de un Autoencoder. De acuerdo con la literatura disponible [20], el número de nodos en la capa oculta debe ser menor que el tamaño de la entrada receptora para aprender la representación comprimida de los datos de entrada.

Otra métrica que es importante considerar es el costo computacional del proceso de entrenamiento, que es el tiempo requerido para entrenar una red neuronal. El número de capas

ocultas y la cantidad de nodos en cada capa oculta afectan el costo computacional promedio de entrenamiento de la red. La red con la mayor cantidad de capas ocultas y nodos en cada capa oculta tendrá el costo computacional promedio de entrenamiento más alto, ya que tendrá más parámetros de la red para entrenar.

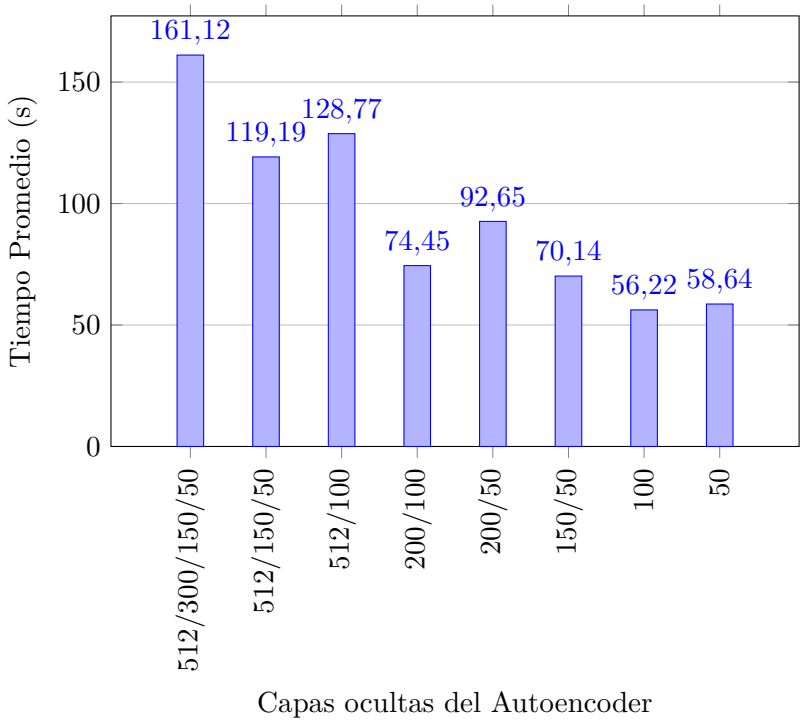


Figura 3.2: Efecto de capas ocultas y nodos en tiempo de ejecución

La Figura 3.2 muestra el costo computacional promedio de entrenamiento para diferentes estructuras de redes consideradas en este proyecto. Se puede observar que el costo computacional promedio de entrenamiento es alto para redes con arquitectura compleja. Se puede observar de la Figura 3.2 que la red con cuatro capas ocultas 500, 300, 150 y 50 nodos en cada una de estas cuatro capas tiene el costo computacional promedio de entrenamiento más alto, mientras que el costo computacional promedio de entrenamiento es reducido cuando la red tiene una arquitectura simple, es decir, con menos capas ocultas y menos nodos en cada capa oculta (i.e., 100/50). De estas observaciones se puede concluir que la adición de nodos en las capas ocultas de la red aumenta la complejidad de la estructura de la red, por lo que requiere más tiempo de ejecución.

Capas Ocultas	Precisión Máxima (%)	Precisión Media (%)	Tiempo Medio (s)
512/300/150/50	92.50	90.78	161.12
512/150/50	94.89	93.67	119.19
512/100	95.65	93.87	128.77
200/100	93.48	92.54	74.45
200/50	93.80	92.17	92.65
150/50	92.39	92.00	70.14
100	88.80	88.41	56.22
50	87.28	85.59	58.64

Tabla 3.1: Resultados de diferentes estructuras de redes

En cuanto a la precisión indicada en la Tabla 3.1, se corrieron cinco entrenamientos para cada topología de red a fin evaluar varias veces la respuesta. A pesar de tener más capas, la red de 500-300-150-50 no se vio beneficiada de la mayor cantidad de nodos. Como se comentó antes, no hay una regla definida para determinar la cantidad de capas y nodos ocultos de una red, lo cual se evidencia en este ejemplo.

Observamos a continuación el resultado de las curvas de las perdidas de entrenamiento de la red:

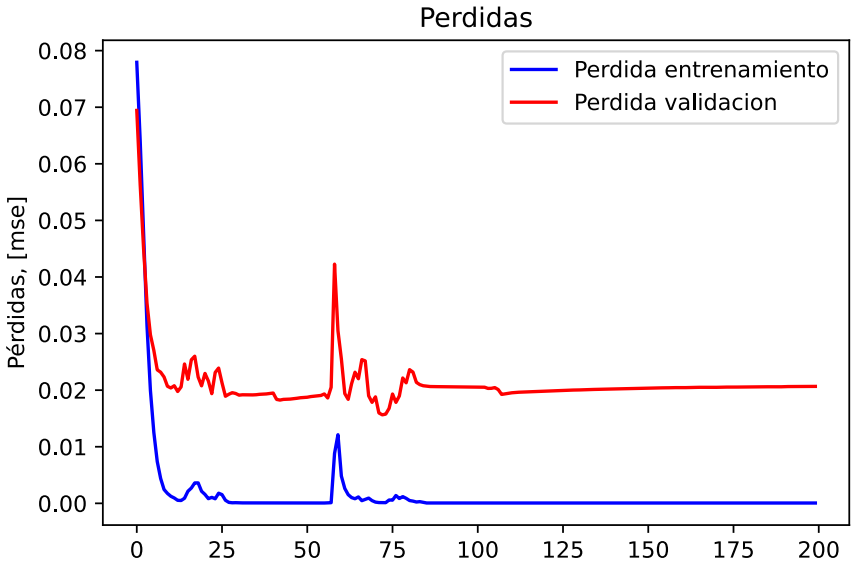


Figura 3.3: Curvas de perdidas de entrenamiento de la red para el dataset de CWRU

La forma y dinámica de una curva de aprendizaje puede ser utilizada para diagnosticar el comportamiento de un modelo de aprendizaje y, en su caso, puede sugerir que tipos de cambios de configuración que se pueden hacer para mejorar el aprendizaje y/o rendimiento.

Un buen fit es el objetivo del algoritmo de aprendizaje y existe entre un modelo con overfit y un modelo underfit. Un buen fit es identificado por una perdida de entrenamiento y validación que disminuye a un punto de estabilidad con un mínimo de una diferencia entre los dos valores finales de perdida. La perdida del modelo generalmente será más baja en el conjunto de datos de

entrenamiento que en el conjunto de datos de validación. Esto significa que esperamos que haya una diferencia entre la perdida de entrenamiento y la perdida de validación. Esta diferencia se llama la ”diferencia de generalización”.

Podemos analizar también la matriz de confusión obtenida. Una Matriz de confusión es una matriz de NxN usada para evaluar la precisión de un modelo de aprendizaje. La matriz de confusión tiene N filas y N columnas, donde cada elemento de la matriz es una representación de la cantidad de elementos correctos e incorrectos para cada clase.

Para un problema de clasificación binaria, tendríamos una matriz de 2x2 como se muestra en la figura:

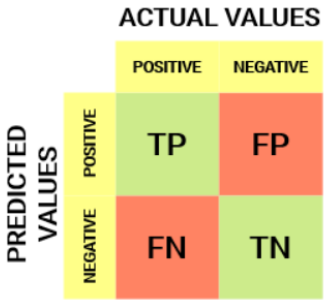


Figura 3.4: Matriz de confusión para un problema de clasificación binaria

La matriz binaria posee:

- La variable objetivo tiene dos valores: Positivo o Negativo.
- La columna representa la clase objetiva.
- La fila representa el valor predicho de la clase objetivo.

Para nuestro caso, luego de correr el algoritmo y la validación obtenemos la siguiente matriz de confusión:

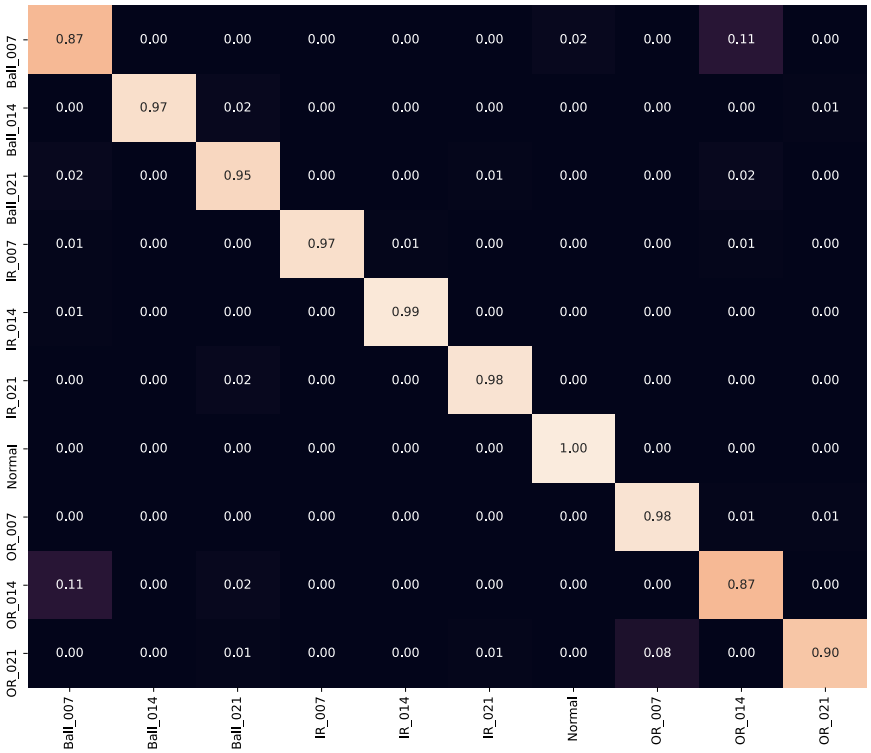


Figura 3.5: Matriz de confusión para el dataset de CWRU

En la matriz de la Figura 3.5 observamos una matriz de 10x10 normalizada donde tenemos las clases objetivos (las distintas fallas) y el valor predicho por la red. Se obtiene un buen resultado en la mayoría de las clases elegidas. Un mal resultado indicaría que la red no es capaz de discernir entre dos clases distintas lo cual podráa indicar una gran correlación entre los datos de ambas clases o que la red no es capaz de distinguir entre las mismas. La diferencia en los valores esperados y predichos se explica con la perdida en la precisión, que no es exactamente 100 % por lo que es esperable que algunas predicciones no correspondan con su label esperado.

Como se nombró anteriormente, la Máxima Correntropía juega un papel importante a la hora de diseñar el Autoencoder para nuestra aplicación. En [12] para el diseño de algunos parámetros del Autoencoder utilizan un algoritmo de optimización llamado AFSA (Artificial Fish Swarm Algorithm) el cual es un algoritmo popular de optimización que puede encontrar máximos globales dentro de ciertos parámetros del algoritmo [21].

Uno de los inconvenientes encontrados en [12] fue que el paper no indica información suficiente para poder replicar las pruebas necesarias de AFSA con el objetivo de optimizar los parámetros claves del Autoencoder. Debido a esta limitación primero se realizó una validación experimental utilizando los mismos valores indicados en la investigación, pero no se lograron buenos resultados, al contrario, los resultados empeoraron bastante. Para ello fue necesario correr algunas pruebas para obtener valores de una manera más experimental.

Planteamos primero el diseño para este proyecto de la fórmula de Correntropía para nuestro

trabajo en el framework de Keras:

```
tf_2pi = tf.constant(2*np.pi, dtype=tf.float32)

@tf.function
def mercer_kernel(x, sigma=0.27):
    return (1 / (tf.sqrt(tf_2pi) * sigma)) * tf.exp((-(x * x) / (2 * sigma
    * sigma)))

@tf.function
def correntropy(y_true, y_pred):
    return -tf.math.reduce_mean(mercer_kernel(y_true - y_pred))
```

Donde uno de los parámetros es el sigma(σ), valor objetivo a optimizar. Planteamos además las funciones de Sparsity del Autencoder (Ecuación 2.5):

```
from keras import backend as K

def kl_divergence(rho, rho_hat):
    return rho * tf.math.log(rho) - rho * tf.math.log(rho_hat) + (1 - rho)
    * tf.math.log(1 - rho) - (1 - rho) * tf.math.log(1 - rho_hat)


class SparseRegularizer(keras.regularizers.Regularizer):

    def __init__(self, rho=0.2, sparsityBeta=1):
        self.rho = rho
        self.sparsityBeta = sparsityBeta

    def __call__(self, x):
        regularization = 0

        activation = tf.nn.sigmoid(x)
        rho_hat = K.mean(activation, axis=0)
        regularization += self.sparsityBeta * K.sum(kl_divergence(self.rho,
    rho_hat))

        return regularization

    def get_config(self):
        return {"name": self.__class__.__name__}

```

También podemos definir la función de Weight Decay (Ecuación 3.4) que se utiliza en el Autoencoder:

```
class WeightDecayRegularizer(keras.regularizers.Regularizer):

    def __init__(self, strength):
        self.strength = strength

    def __call__(self, x):
        return self.strength * tf.reduce_sum(tf.square(x))

    def get_config(self):
```

```
10          return {'strength': self.strength}
11
```

Planteadas las funciones, se procedió a evaluar el impacto de distintos tamaños de Kernel (Ecuación 3.2) en la precisión de la red. Para ello se tomaron 20 valores distintos haciendo foco en el valor de σ donde mejor se comporta la Correntropía, que está demostrado que se encuentra en el rango de valores [0,2, 2] [22].

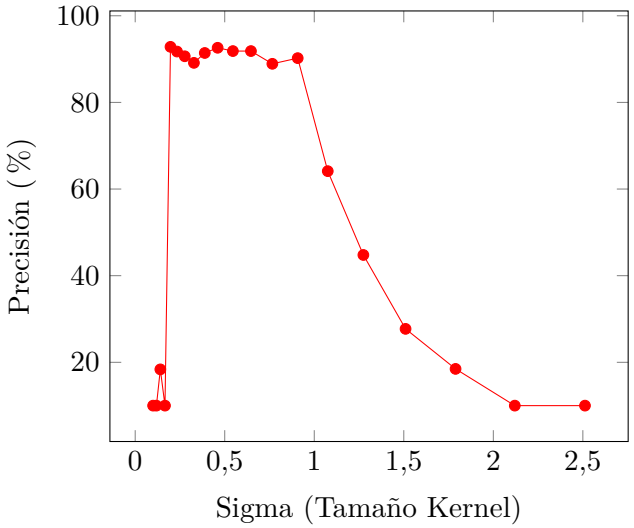


Figura 3.6: Comparación tamaño Kernel Autoencoder vs Precisión

Donde se observa en la Figura 3.6 que para nuestra aplicación el tamaño de Kernel σ se encuentra en los valores [0,2, 1] con precisión máxima del 93 %-94.67 % y luego se empieza a notar una caída en la precisión de la red, por lo que se toma un valor de $\sigma = 0{,}27$ para el Autoencoder propuesto.

4 Validación Experimental

4.1. Primera Etapa

Para realizar la validación de la red utilizada en este proyecto y debido a las limitaciones de equipamiento disponible, se construyó un banco de ensayos experimental el cual consta de un motor de hormigonera DAF 990002E 1HP y 1500 RPM, una base de madera amortiguada con soportes de goma, un eje de acero 1045 con kit de rulemanes autocentrantes UCP 204 de la marca FK Bearing Group, correa BTS y un acelerómetro ADXL 357 de Analog Devices, sumado una placa FPGA Zybo Z7-10 de Xilinx para el procesamiento.

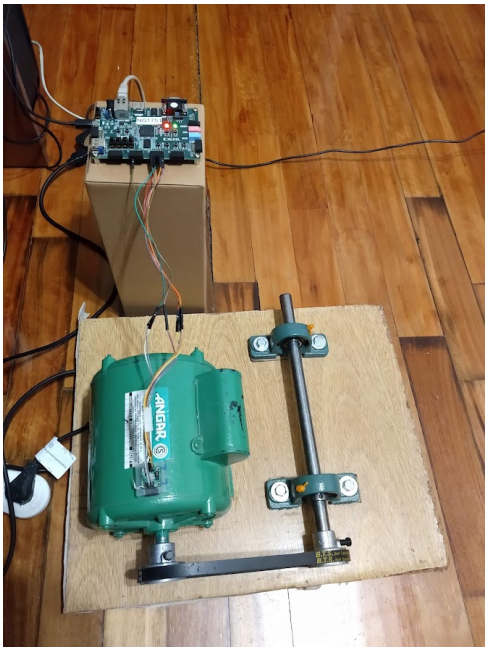


Figura 4.1: Banco de ensayos experimental para la validación de la red

Podemos observar un diagrama en bloques de la configuración final del experimento:

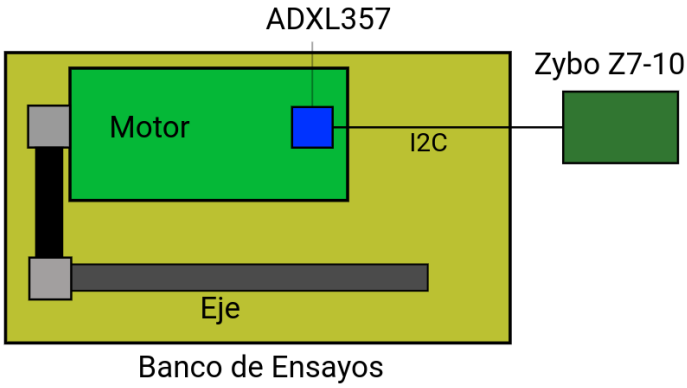


Figura 4.2: Diagrama en bloques del experimento

En la Figura 4.2 el banco de ensayos es medido con el acelerómetro ADXL357 mediante un montaje realizado magnéticamente en el motor. Luego el sensor envía los datos mediante el protocolo I2C a la placa FPGA Zybo Z7-10 la cual procesa los datos recibidos y realiza la inferencia de la red neuronal. Luego, se utiliza una PC como terminal de visualización de los datos obtenidos de la inferencia de la placa.

Para el experimento se realizaron cinco tipos de condiciones de funcionamiento distintas, las cuales son:

- **Condición Normal**: El equipamiento funcionando correctamente
- **Desbalanceo de Carga**: El eje de acero es corrido 3cm para atrás por lo que queda desalineado con el eje del motor.
- **Desbalanceo Frontal**: De los cuatro tornillos que sostienen al motor, los dos tornillos delanteros son levantados 5mm aproximadamente provocando que el motor se encuentre inclinado hacia atrás, provocando un desbalanceo frontal en la correa.
- **Desbalanceo Lateral**: Los dos tornillos laterales del motor se levantan 5mm aproximadamente provocando que el motor esté inclinado hacia un costado, generando un desbalanceo lateral en la correa.
- **Falla en la Carga**: Utilizando un elemento externo se genera un roce en el eje secundario de acero provocando una vibración externa al sistema.

Para cada condición de operación se tomaron 65536 muestras con el acelerómetro ADXL357 desde la FPGA ZyboZ7 los cuales se almacenaron y procesaron dividiendo los mismos samples de 512 muestras cada uno teniendo un total de 128 samples, 88 para entrenamiento y 40 para validación/testing.

Condición de Falla	**Muestras por Sample**	**Samples Entrenamiento**	**Samples Testing**	**Total**
Normal	512	88	40	65536
Desbalanceo de Carga	512	88	40	65536
Desbalanceo Frontal	512	88	40	65536
Desbalanceo Lateral	512	88	40	65536
Falla de la Carga	512	88	40	65536

Tabla 4.1: Detalle de muestras adquiridas para el primer experimento

Dado que el set de datos del experimento es distinto, podemos realizar el mismo análisis que para el set de datos de CWRU a fin de evaluar una topología que se adapte a nuestro problema de una mejor manera, ya que el input utilizado es distinto con tal de asimilarse al utilizado en [12].

Capas Ocultas	**Precisión Maxima (%)**	**Precisión Media (%)**	**Tiempo Medio (s)**
300/150/50	93.59	90.78	42.77
300/100	90.47	89.3	39.09
300/50	91.25	89.53	38.32
200/50	90.08	89.38	36.28
150/50	90.86	88.98	36.7
100	89.3	88.36	36.32
50	87.73	86.25	35.37

Tabla 4.2: Resultados de diferentes estructuras de redes

Podemos observar la matriz de confusión para los datos utilizados:

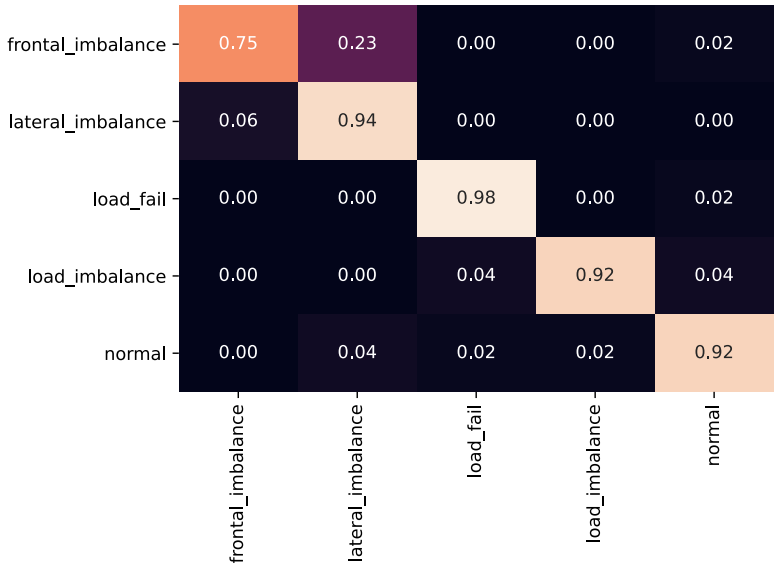


Figura 4.3: Matriz de confusión para el banco de ensayos experimental

En la Figura 4.3 podemos observar una matriz de 5x5 normalizada donde tenemos los labels customizados para el set de datos propio: *frontal_imbalance*, *lateral_imbalance*, *load_fail*, *load_imbalance* y *normal* y una precisión media del modelo de 92.35 %. En la matriz vemos que algunas categorías son mejor reconocidas que otras siendo la menos precisa la categoría *frontal_imbalance* vs *lateral_imbalance* donde la conclusión a la baja diferencia es debido a la forma de reproducción de la condición de funcionamiento.

4.2. Profiling de la Red

Si bien se obtuvieron resultados para el set de datos experimental, el proyecto se enfoca en la implementación de la red neuronal en una FPGA. Para ello es necesario realizar el proceso de cuantización de datos así como adaptar la red para ajustarse a los recursos disponibles de la misma.

Es necesario primero determinar el rango de los pesos de la red, para que en el proceso de cuantización no elegir un tipo de datos que trunque el valor de los pesos o biases de la red haciendo que se pierda información necesaria para el procesamiento.

Para el análisis se utiliza un diagrama de caja (box plot) que es un método estandarizado gráficamente que muestra una serie de datos numéricos a través de sus cuartiles. De esta manera se observan a simple vista la mediana y los limites máximos y mínimos de la serie de datos a analizar.

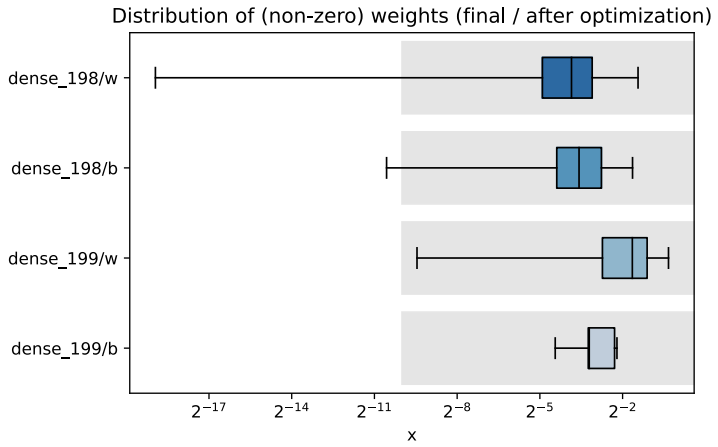


Figura 4.4: Box plot para el set de datos experimental

La parte gris del gráfico de la Figura 4.4 indica el rango en el cual se puede parametrizar la configuración de hls4ml. Se busca que la cuantización cubra estos valores y luego se puede ir ajustando de manera experimental para lograr una mejor optimización en cuanto a recursos o latencia según sea el caso.

Debido a la forma de trabajo de las FPGA es necesario también evaluar los recursos que la implementación de la red pueda llegar a requerir. Para ello uno de los puntos a analizar es el input de la red, ya que cada entrada necesaria consume recursos de la FPGA. Para el análisis experimental se utilizó una entrada de 512 datos y luego la red posee una serie de capas ocultas con una salida de 5 datos (uno por cada condición de funcionamiento).

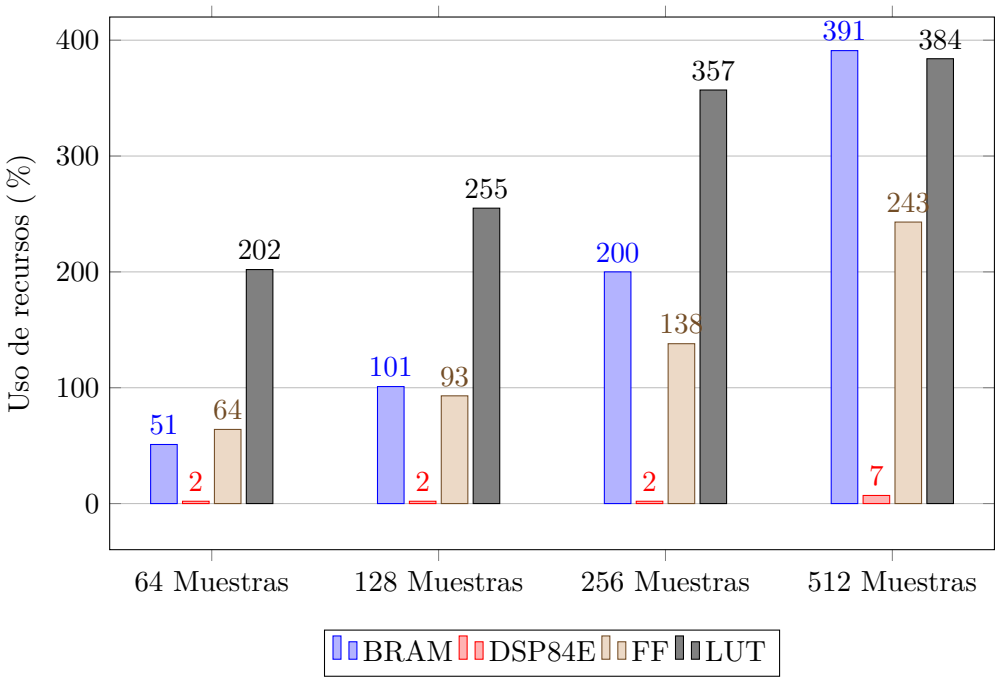


Figura 4.5: Comparación de uso de recursos segun input de la red

En la Figura 4.5 se puede observar como es imposible implementar una red de input 512 para la Zybo Z7-10, ya que los recursos estimados superan ampliamente los recursos disponibles. A pesar de que en el proceso de síntesis se realizan optimizaciones en el uso de recursos, no es posible implementar y optimizar una red de tal dimensión. Para ello se adapta la red para tener un input de 64 muestras la cual si puede ser optimizada y sintetizada para trabajar en la placa elegida.

Al tratarse de una placa de bajos recursos es necesario restringir la arquitectura de la red para adaptarse a la misma. Así como se comparó el efecto del tamaño de la red para distintos inputs, podemos analizar para un input fijo de 256 datos que sucede al trabajar en tres placas FPGA: bajos recursos, recursos medios y altos recursos:

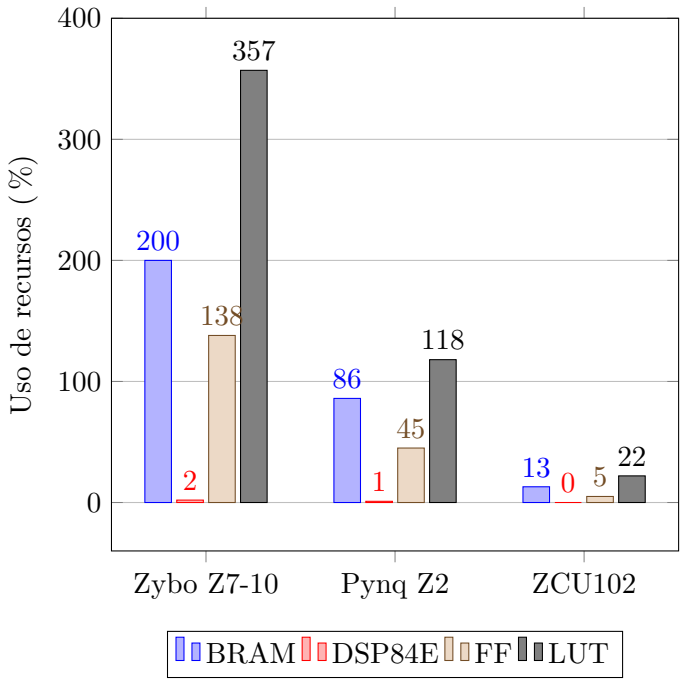


Figura 4.6: Comparación de uso de recursos para distintos modelos de FPGA

Se hace evidente en la Figura 4.6 el impacto en los recursos para las tres placas del ejemplo. Para la ZCU102 el uso de recursos es mínimo, por lo que de tener disponibilidad se podría llegar a implementar una arquitectura de red neuronal más grande, obteniendo beneficios en la inferencia. Sin embargo, para la Zybo Z7-10 se requieren más recursos que la ZCU102, por lo que no es posible implementar una red de tal dimensión y como vimos antes fue necesario reducir considerablemente el tamaño de la red para poder adaptarla a la placa.

La red final de la implementación utilizada y con una obtención experimental consta de 64 entradas, una capa oculta de 40 neuronas y la salida de la red de 5 neuronas (una por cada condición de funcionamiento). El set de datos se adapta de la siguiente manera:

Condición de Falla	**Muestras por Sample**	**Samples Entrenamiento**	**Samples Testing**	**Total**
Normal	64	920	104	65536
Desbalanceo de Carga	64	920	104	65536
Desbalanceo Frontal	64	920	104	65536
Desbalanceo Lateral	64	920	104	65536
Falla de la Carga	64	920	104	65536

Tabla 4.3: Detalle de muestras ajustadas para el primer experimento en FPGA

4.3. Cuantización

Para la cuantización de la red se utiliza la herramienta QKeras que es una extensión de Keras que provee una implementación de las capas de Keras que crean parámetros y capas de activación, y realiza operaciones aritméticas, para que se pueda crear rápidamente una red cuantizada basándonos en la parametrización elegida.

Se eligen como parámetros de datos de punto fijo con una longitud de 2 bits para la parte entera y 10 bits para la parte fraccionaria con base en lo analizado en el profiling de la red. Se agrega además un pruning de la red del 75 %. Las capas de la red quedan conformadas como 64->40->5.

Int Bits	**N Bits**	**Precisión Media (%)**
2	8	93.26
2	10	94.48
2	12	94.28
2	14	93.13
2	16	93.6

Tabla 4.4: Comparativa tamaño de bits

Con la reducción de la red implementada, se logra una precisión en la inferencia máxima de **96.09 %**. Es necesario recordar que el proyecto se basa en ofrecer más información para un posterior análisis del estado de la máquina en conjunto con otros parámetros de mantenimiento predictivo. En nuestro caso, al reducir la cantidad de muestras y teniendo una red muy pequeña, el % de precisión aumentó un par de puntos, pero de todas formas el resultado es válido para la aplicación de este proyecto, lo que puede presentar diferencias para otras condiciones de funcionamiento.

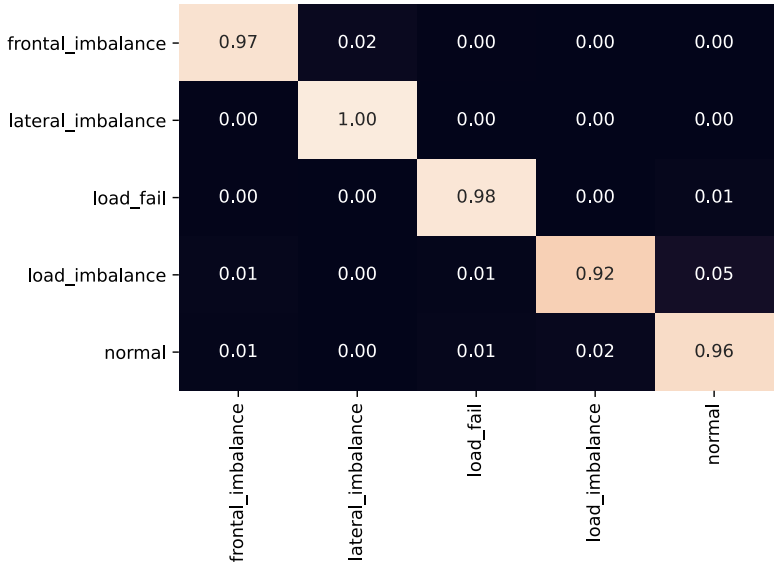


Figura 4.7: Matriz de confusión para la red final de input 64

En la Figura 4.7 podemos observar el resultado de la inferencia de la red. Al ser una tabla normalizada no se nota el aumento en la cantidad de muestras procesadas, pero se observa una mejora en la precisión.

4.4. Validación en FPGA

A fin de utilizar el modelo cuantizado previamente en la FPGA, es necesario convertir el modelo de *QKeras* en un modelo HLS y luego convertirlo y programarlo en la FPGA utilizando Vivado como nombramos anteriormente. Recordemos que la red sintetizada con Vivado HLS genera un bloque IP el cual podemos observar en la siguiente imagen:

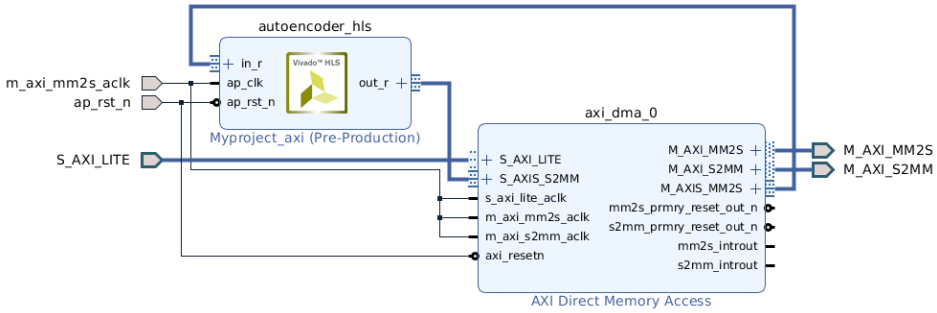


Figura 4.8: Bloque IP del autoencoder

Este bloque IP luego debe ser integrado con otros componentes para poder funcionar en la FPGA objetivo. Para ello realizamos el siguiente conexionado en Vivado:

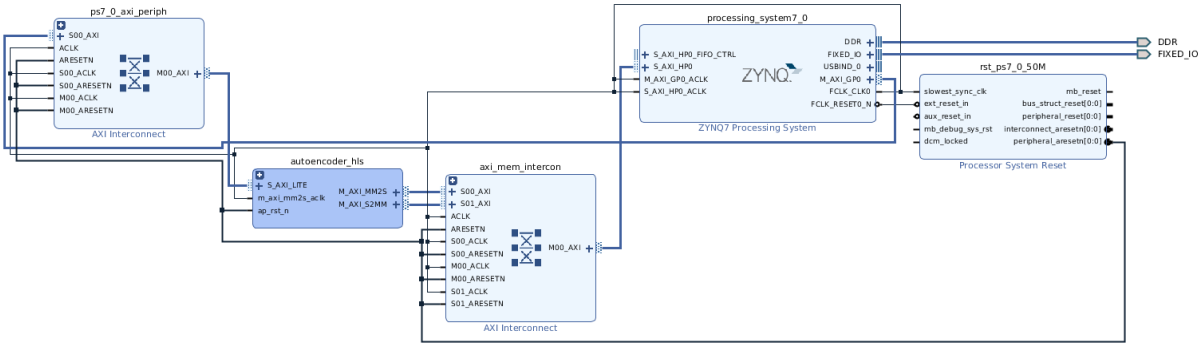


Figura 4.9: Circuito final en Vivado

Donde en la Figura 4.9 observamos distintos bloques de conexión para distintos componentes de la FPGA y dentro del bloque *autoencoder_hls* se encuentra el bloque IP de la Figura 4.8.

Planteado el circuito, primero debemos realizar algunas optimizaciones para poder realizar la implementación del bloque IP en la FPGA. Una de las parametrizaciones que permite hls4ml (Figura 2.11) es el término **Reuse Factor** el cual controla el pipeline de datos a ser utilizado por la FPGA. Un bajo número de reúso de recursos permite una mayor paralelización a costo de un mayor uso de recursos, mientras que un bajo valor de reúso permite una mayor optimización de recursos a coste de una mayor latencia.

Corrida la conversión podemos observar una comparativa entre el resultado obtenido en Qkeras, hls4ml previo a la síntesis de hardware y el obtenido luego de la síntesis con Vivado.

Modelo	Precisión (%)
QKeras	96.48
hls4ml	88.67
Vivado HLS	88.67

Tabla 4.5: Comparativa de resultados de inferencia con hls4ml/Vivado

Donde observamos en la Tabla 4.5 que perdemos alrededor del 8 % de precisión en la conversión a hls4ml para nuestro modelo debido a limitaciones en la optimización para nuestra placa.

QKeras	HLS4ML	HLS Vivado	Reuse Factor
96.48	88.67	88.67	2
96.48	88.67	88.67	4
96.48	88.67	88.67	8
96.48	88.67	88.67	16
96.48	88.67	88.67	32
96.48	88.67	88.67	64
96.48	88.67	88.67	128
96.48	88.67	88.67	256
96.48	88.67	88.67	320
96.48	88.67	88.67	512
96.48	88.67	88.67	640
96.48	88.67	88.67	1280

Tabla 4.6: Comparativa tamaño de Reuse Factor

En la Tabla 4.6 podemos observar que la precisión no depende del valor de reuse factor, ya que como habíamos nombrado impacta en la latencia de la FPGA y el uso de recursos. Se detalla en la siguiente Tabla 4.7 un ejemplo del informe de uso de recursos que nos provee Vivado al realizar la síntesis de la red:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	40	13818	-
FIFO	-	-	-	-	-
Instance	63	2	17825	21257	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	224	-
Register	0	-	4940	352	-
Total	**63**	**2**	**22805**	**35651**	**0**
Available	**120**	**80**	**35200**	**17600**	**0**
Utilization (%)	**52**	**2**	**64**	**202**	**0**

Tabla 4.7: Reporte de uso de recursos de Vivado

Si ahora repetimos el proceso de conversión y síntesis del modelo, pero analizando los recursos

utilizados en vez de la precisión como en la Tabla 4.6, podemos analizar el impacto del parámetro **Reuse Factor** en el uso de recursos de la placa Zybo Z7-10 que se utiliza para este desarrollo:

REUSE FACTOR	BRAM_18K	DSP48E	FF	LUT	LATENCY
2	431	431	779	1538	34
4	240	240	402	807	36
8	144	144	213	434	42
16	95	95	117	248	60
32	71	71	71	151	95
64	60	60	50	110	127
128	54	54	49	116	352
256	51	51	48	114	481
320	50	50	47	113	544
512	50	50	47	112	736
640	51	51	47	113	865
1280	51	51	47	112	1504

Tabla 4.8: Comparación uso de recursos (%) vs Reuse Factor

Y, expresando los mismos resultados de manera gráfica, podemos visualizar que el uso de recursos cae a medida que aumentamos el valor de Reuse Factor donde llega un punto que por más que se incremente el factor de reúso no se observa una mejora en el uso de recursos:

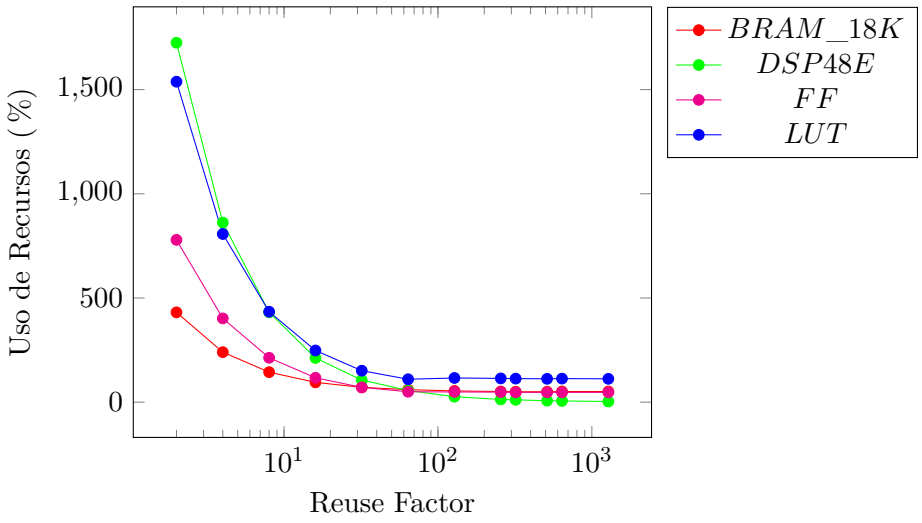


Figura 4.10: Comparación de uso de recursos vs Reuse Factor

Por el contrario, podemos visualizar en la Figura 4.11 lo que sucede con la latencia del bloque IP generado al aumentar el factor de reúso si tomamos como unidad la cantidad de ciclos de clock necesarios para realizar la inferencia de los datos adquiridos:

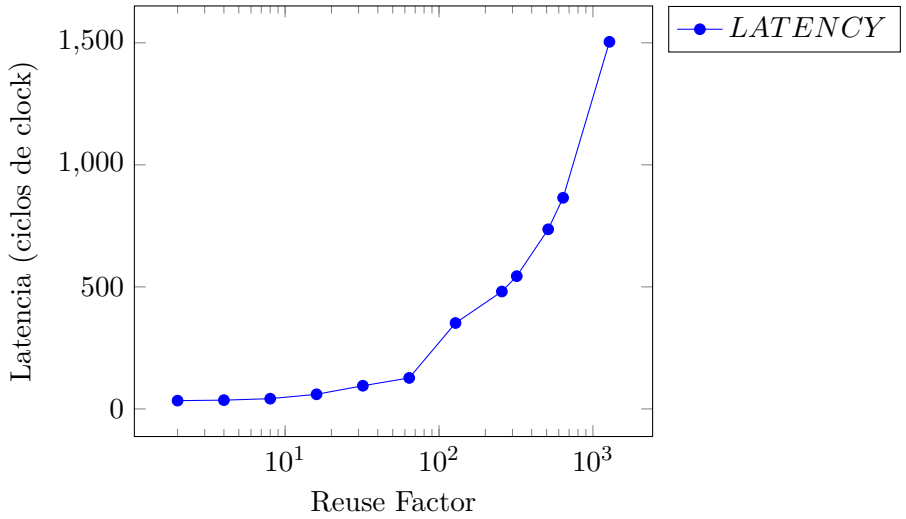


Figura 4.11: Comparación de Latencia vs Reuse Factor

Se observa un trade-off entre aumentar el Reuse Factor para disminuir la cantidad de recursos necesarios para implementar la red en una FPGA de bajos recursos y el aumento de la latencia para realizar el procesamiento. Para aplicaciones donde el timming sea crítico, un valor de Reuse Factor alto no sería recomendable por su aumento en la latencia y por lo pronto es recomendable la inversión en una FPGA de mayores recursos a fin de mantener la latencia baja. Para nuestro proyecto, la latencia no es un factor considerado (aunque si evaluado), por lo que el aumento del factor de reúso no es relevante pero si necesario para poder implementar la red necesaria en la FPGA, por lo que se elige un $Reuse\_Factor = 64$ el cual permite la implementación de la red para los recursos disponibles.

5 Resultados

5.1. Resultados de la implementación

Una vez generado y cargado el modelo en la placa, utilizaremos python a través de PYNQ para poder correr la inferencia de la red. Se diseñó la siguiente clase para el análisis de los resultados:

```
class NeuralNetworkOverlay(Overlay):
    def __init__(self, bitfile_name, dtbo=None, download=True, ignore_version=False, device=None):
        super().__init__(bitfile_name, dtbo=dtbo, download=download, ignore_version=ignore_version, device=device)

    def _print_dt(self, timea, timeb, N):
        dt = (timeb - timea)
        dts = dt.seconds + dt.microseconds * 10**-6
        rate = N / dts
        print("Classified {} samples in {:.2f} seconds ({:.2f} inferences / s)".format(N, dts, rate))
        return dts, rate

    def predict(self, X, y_shape, dtype=np.float32, debug=None, profile=False, encode=None, decode=None):
        if encode is not None:
            X = encode(X)
        with allocate(shape=X.shape, dtype=dtype) as input_buffer, \
            allocate(shape=y_shape, dtype=dtype) as output_buffer:
            input_buffer[:] = X

            if profile:
                timea = datetime.now()

            t = time.time()
            self.hier_0.axi_dma_0.sendchannel.transfer(input_buffer)
            self.hier_0.axi_dma_0.recvchannel.transfer(output_buffer)
            self.hier_0.axi_dma_0.sendchannel.wait()
            self.hier_0.axi_dma_0.recvchannel.wait()
            fast_time = time.time() - t
            timeb = datetime.now()

            result = output_buffer.copy()
        if decode is not None:
            result = decode(result)
        if profile:
            dts, rate = self._print_dt(timea, timeb, len(X))
            return result, dts, rate, fast_time
        return result

```

Y para realizar la inferencia, instanciamos la clase y le pasamos como input de datos los archivos *.npz* que contienen los datos generados en el banco de ensayos y el archivo *bitfile* que contiene el modelo sintetizado en Vivado:

```
NN = NeuralNetworkOverlay(bitfile_name="./hls_dma_zyboz7.bit")

muestras_por_segundo = []
for _ in range(10):

    y_pynq, dts, rate, fast_time = NN.predict(X=X, y_shape=(X.shape[0],
    y_hls.shape[1]), dtype=np.float32, debug=False, profile=True, encode=
    None, decode=None)

    muestras_por_segundo.append(fast_time/X.shape[0])
assert((y_pynq == y_hls).all())

#Classified 512 samples in 0.04 seconds (11827.21 inferences / s)
#Classified 512 samples in 0.04 seconds (11826.12 inferences / s)
#Classified 512 samples in 0.04 seconds (11903.38 inferences / s)
#Classified 512 samples in 0.04 seconds (11899.78 inferences / s)
#Classified 512 samples in 0.04 seconds (11905.87 inferences / s)
#Classified 512 samples in 0.05 seconds (10580.48 inferences / s)
#Classified 512 samples in 0.04 seconds (11854.60 inferences / s)
#Classified 512 samples in 0.04 seconds (11884.87 inferences / s)
#Classified 512 samples in 0.04 seconds (11860.64 inferences / s)
#Classified 512 samples in 0.04 seconds (11924.45 inferences / s)

```

Observamos que la red procesa de manera muy rápida los datos al estar acelerada por hardware y además con la línea *y_pynq == y_hls* nos aseguramos que el resultado obtenido en la FGPA es exactamente el mismo que el obtenido en la simulación de HLS previo a la síntesis, por lo que validamos el flujo de desarrollo de HLS y hls4ml para la generación de redes neuronales en FPGAs.

El proceso nombrado solamente comprueba la funcionalidad de la red utilizada utilizando los datos obtenidos del banco de ensayos, pero no verifica la funcionalidad cuando se toman datos en tiempo real dado que la validación se utiliza con un segmento de los datos obtenidos originalmente. Si bien es común el uso de datos de esta manera, en caso de que la condición de funcionamiento cambie no podemos verificar que la red se adapte a los nuevos datos.

Para ello necesitamos generar un script que corra en la FPGA y se encargue de obtener datos y procesarlos en tiempo real a fin de evaluar el comportamiento de la red neuronal con nuevos datos de prueba. El script generado contiene el siguiente flujo de datos:

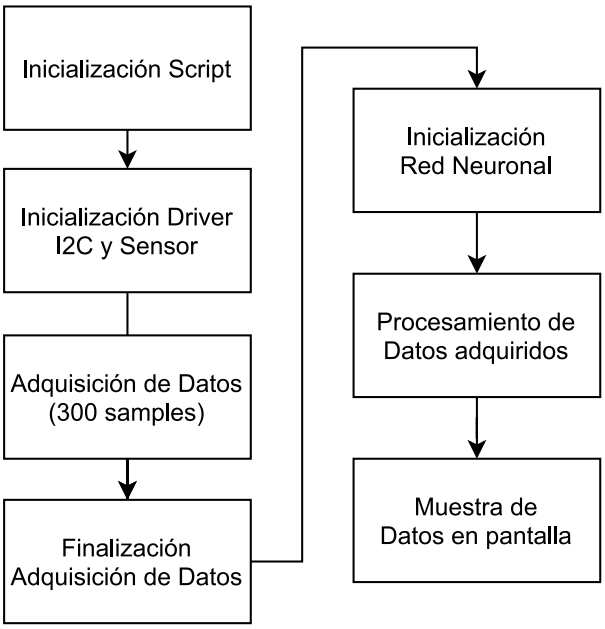


Figura 5.1: Script de adquisición de datos en tiempo real en la FPGA

Donde se toma una cierta cantidad de datos respetando el input de la red y se dividen en distintos samples para que la red posea un flujo de datos considerable. Luego los datos obtenidos por el sensor son procesados por la red y se muestra un resultado en pantalla.

Para validar el script, se reprodujeron las condiciones iniciales de funcionamiento y se corrió el script para obtener nuevos resultados:

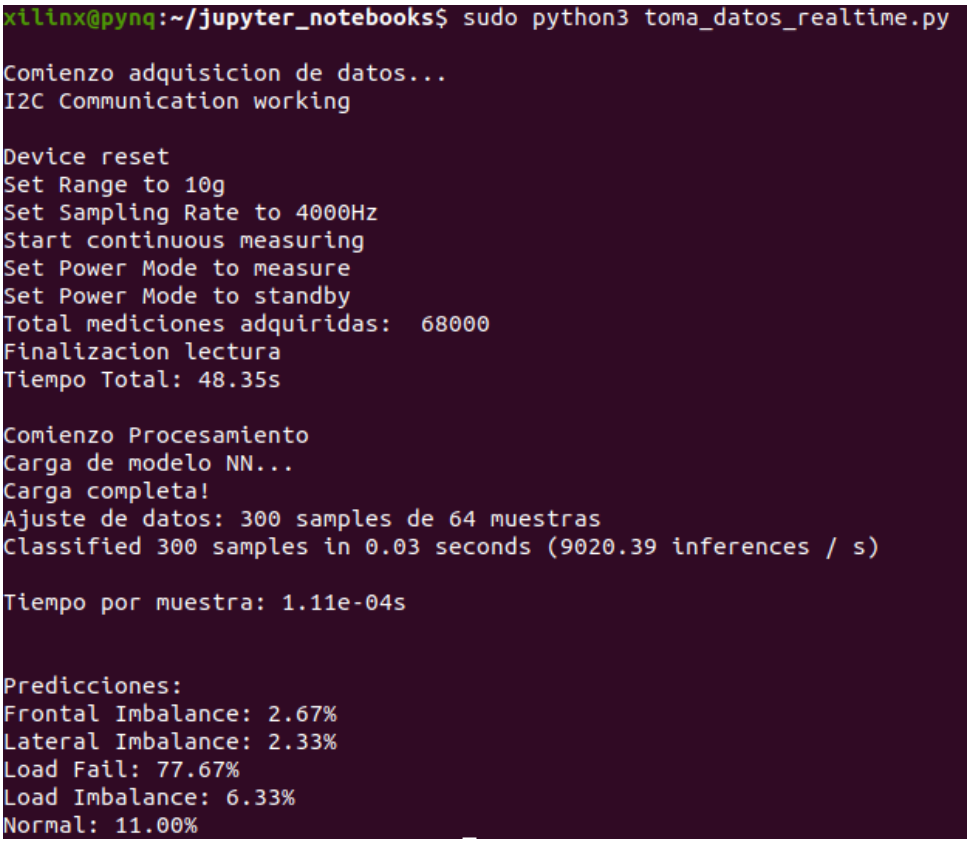


Figura 5.2: Resultados script real time en FGPA

A pesar de ejecutar la red en tiempo real y de correr las pruebas varias veces, no se obtuvieron resultados tan buenos como cuando se utilizaron los datos obtenidos en el ensayo inicial. Si bien sabemos que la red funciona, el uso de un banco de ensayos experimental de bajo costo produjo que algunas condiciones de funcionamiento no pudieran ser replicadas con exactitud, por lo que los datos obtenidos en los nuevos experimentos y debido a eso, la precisión de la red es menor a la detectada con los datos originales como se observa en la Figura 5.2. Este resultado es esperable, ya que para lograr una red muy abarcativa y precisa es necesario que el tamaño de la red sea más grande para poder aprender más features de los datos y así a pesar de tener distintos datos de entrada, detectar con mejor facilidad las variaciones en los mismos. Incluso hoy en día, muchas redes de gran tamaño tienen dificultades al detectar variaciones significativas en datos de entrada para los cuales no fueron entrenadas, por lo que gran parte del esfuerzo de entrenamiento de redes neuronales es proveer a la red con variaciones de datos de entrada con técnicas como data augmentation para que pueda aprender de manera más efectiva. Para este proyecto, este tipo de técnicas no es factible dado que la red necesita aprender ciertas condiciones de funcionamiento particulares.

5.2. Análisis de sensibilidad de la red

A fin de verificar la sensibilidad de la red, evaluar posibles aplicaciones y lograr condiciones de funcionamiento más repetibles/reproducibles se realizó un nuevo experimento con el banco

de ensayos. Para este experimento se generaron cinco nuevas condiciones de funcionamiento en el motor. Para el experimento se desconectaron la correa y la carga a fin de dejar el eje del motor libre y se fueron agregando distintos pesos al eje a fin de provocar distintos comportamientos de funcionamiento.

Con el objetivo de lograr esto, se utilizó la polea conectada al eje del motor y se fueron agregando distintos pesos utilizando tuercas de distintos pesos (controlado) a fin de conocer con exactitud la cantidad de carga agregada en cada condición de funcionamiento y tomando nota para su posterior análisis. Para la medición de los pesos se utilizó una balanza +/-1g por lo que los pesos deben considerarse con algún margen de error.

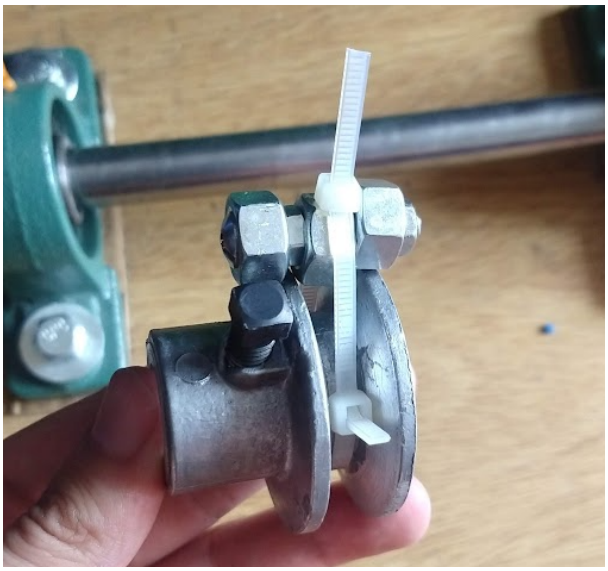


Figura 5.3: Polea con peso

En la Figura 5.3 podemos observar la polea que luego va conectada al eje del motor con el agregado de la carga compuesta por un tornillo y distintas tuercas para variar el peso de la misma. El experimento se repitió varias veces con distinta cantidad de tuercas como lo indica la Tabla 5.1 y para mantener las condiciones de la red neuronal, se adquirieron los siguientes datos:

Condición de Falla	**Muestras por Sample**	**Samples Entrenamiento**	**Samples Testing**	**Total**
Polea Sola (90g)	64	920	104	65536
Polea +9g	64	920	104	65536
Polea +15g	64	920	104	65536
Polea +22g	64	920	104	65536
Polea +30g	64	920	104	65536

Tabla 5.1: Datos de ensayo para la polea con peso

La tabla 5.1 muestra la cantidad de datos adquiridos para cada condición de funcionamiento propuesta para este experimento. Los samples adquiridos son de 64 muestras por sample (para respetar el input de la red desarrollada) y se utilizaron 920 samples para el entrenamiento y 104 samples para el testing. El total de muestras adquiridas es de 65536.

Se mantuvo la configuración de la red 64->40->5, donde 64 es el input del sample de vibraciones, 40 es la capa oculta de 40 nodos y 5 es la capa de salida de 5 nodos, uno por cada condición de funcionamiento: *Polea Sola*, *Polea +9g*, *Polea +15g*, *Polea +22g*, *Polea +30g*. La cuantización se mantiene en 2 bits para la parte entera y 10 bits para la parte fraccionaria. El pruning permanece constante en 75 % y se entrena la red por 50 épocas. Se entrenó la red 5 veces por su característica aleatoria a fin de encontrar el mejor modelo. También se mantiene, para la sintetización de la red, un $Reuse\_Factor = 64$ para lograr la reutilización de recursos necesaria a fin de lograr su implementación en la FPGA Zybo Z7-10.

Podemos evaluar primero el impacto de σ (tamaño de kernel) en la Correntropía para este set de datos y luego podemos correr el entrenamiento de la red para evaluar la precisión máxima obtenida.

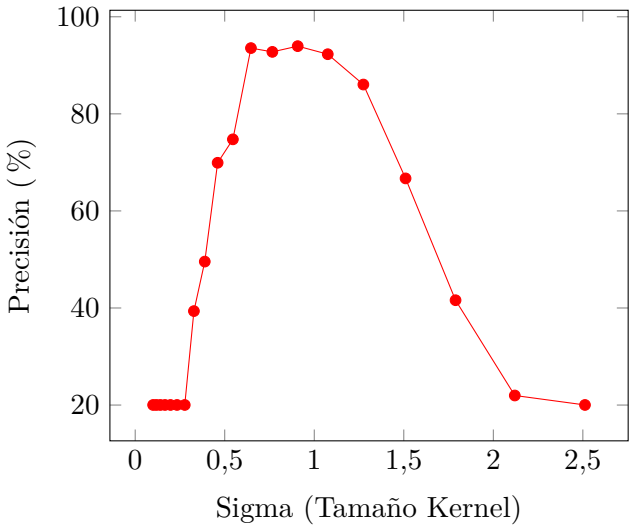


Figura 5.4: Comparación tamaño Kernel Autoencoder vs Precisión

Donde observamos en la Figura 5.4 que para esta configuración de red/datos el valor óptimo es de $\sigma = 0{,}8237$ (tamaño de kernel). Esto se debe a que el cambio en el tamaño del kernel afecta a la función de perdida del Autoencoder y para distintos datos, puede ser mejor un σ diferente tal como observamos en la investigación de referencia [12]. Procedemos luego a utilizar el σ óptimo para entrenar la red.

Iteración	**Precisión (%)**	**Tiempo (s)**
1	91.99	31.81
2	92.29	24.63
3	93.85	23.40
4	93.36	22.78
5	94.82	19.79

Tabla 5.2: Resultados de entrenamiento

De la Tabla 5.2 podemos observar que la precisión media de la red fue de 93.12 % y el tiempo promedio de 24.97s. El mejor modelo se obtuvo en la iteración n°5 con una precisión de 94.82 % y un tiempo de entrenamiento de 19.79s. Quedando entonces con la síntesis en HLS:

- Accuracy QKeras: 94.82 %
- Accuracy HLS4ML: 94.73 %
- HLS_Score: 94.73 %

Podemos además evaluar el desempeño del Autoencoder utilizando la matriz de confusión:

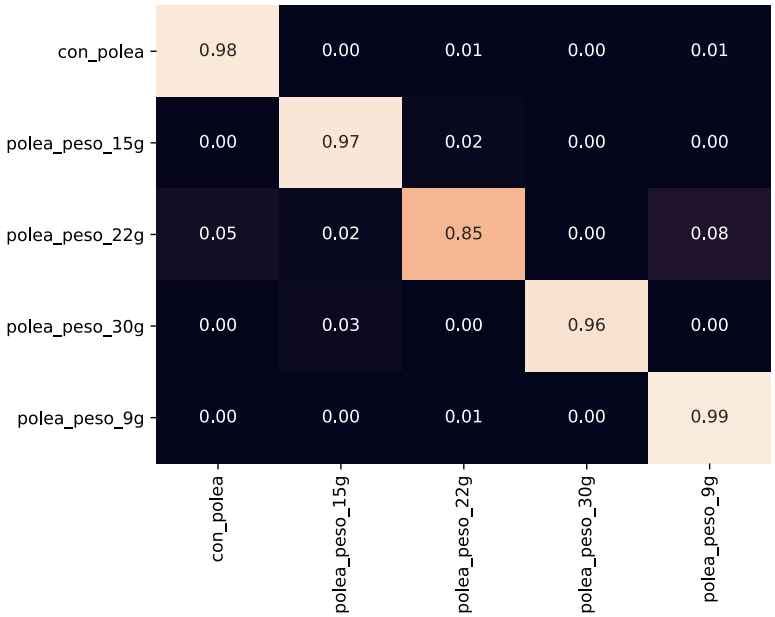


Figura 5.5: Matriz de confusión experimento de polea con pesos

En la matriz de confusión normalizada de la Figura 5.5 observamos las 5 categorías de condiciones de funcionamiento con sus respectivas predicciones. El resultado general fue muy bueno para cuatro de las cinco categorías de condiciones de funcionamiento notando una pequeña caída para Polea +22g.

Si comparamos los pesos de la red, podemos observar el efecto de la cuantización, donde observamos en la Figura 5.6 que los pesos máximos de la red llegan a 2 bits de enteros y 10 para la parte fraccionaria tal como se había configurado.

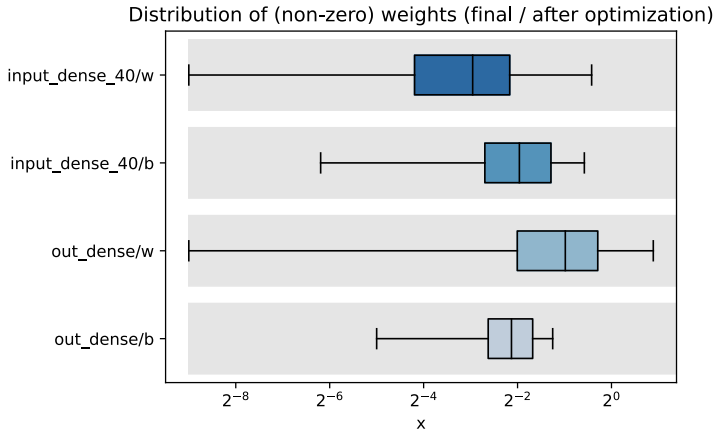


Figura 5.6: Box plot para la red de predicción de pesos

Para proceder, debemos generar nuevamente el archivo *bitfile* con Vivado en el cual se sintetiza el modelo HLS creado y se convierte a un bloque IP el cual luego debemos combinar con otros bloques de Vivado para lograr el circuito necesario para la inferencia de la red.

Una vez grabada la FPGA procedemos a realizar la prueba de funcionamiento de la red. Para ello, tomamos el programa utilizado para el experimento anterior y corremos la inferencia de la red con los datos de prueba.

```
Classified 512 samples in 0.05 seconds (11146.67 inferences / s)
Classified 512 samples in 0.04 seconds (11890.11 inferences / s)
Classified 512 samples in 0.04 seconds (11871.36 inferences / s)
Classified 512 samples in 0.04 seconds (11909.19 inferences / s)
Classified 512 samples in 0.04 seconds (11911.41 inferences / s)
Classified 512 samples in 0.05 seconds (9955.28 inferences / s)
Classified 512 samples in 0.04 seconds (11891.77 inferences / s)
Classified 512 samples in 0.04 seconds (11923.89 inferences / s)
Classified 512 samples in 0.04 seconds (11715.71 inferences / s)
Classified 512 samples in 0.04 seconds (11896.19 inferences / s)

Accuracy QKeras:              94.82%
Accuracy hls_model.predict:   94.73%
Accuracy hls4ml on PYNQ:      94.73%

```

Nuevamente, el resultado obtenido en PYNQ (Zybo Z7-10) no posee perdidas en precisión respecto a la red sintetizada en HLS. Para realizar la última verificación experimental, procedemos a repetir el experimento inicial, pero utilizando la red sintetizada para predecir las distintas condiciones de funcionamiento del experimento. Para ello, corremos nuevamente el script para toma de datos realtime:

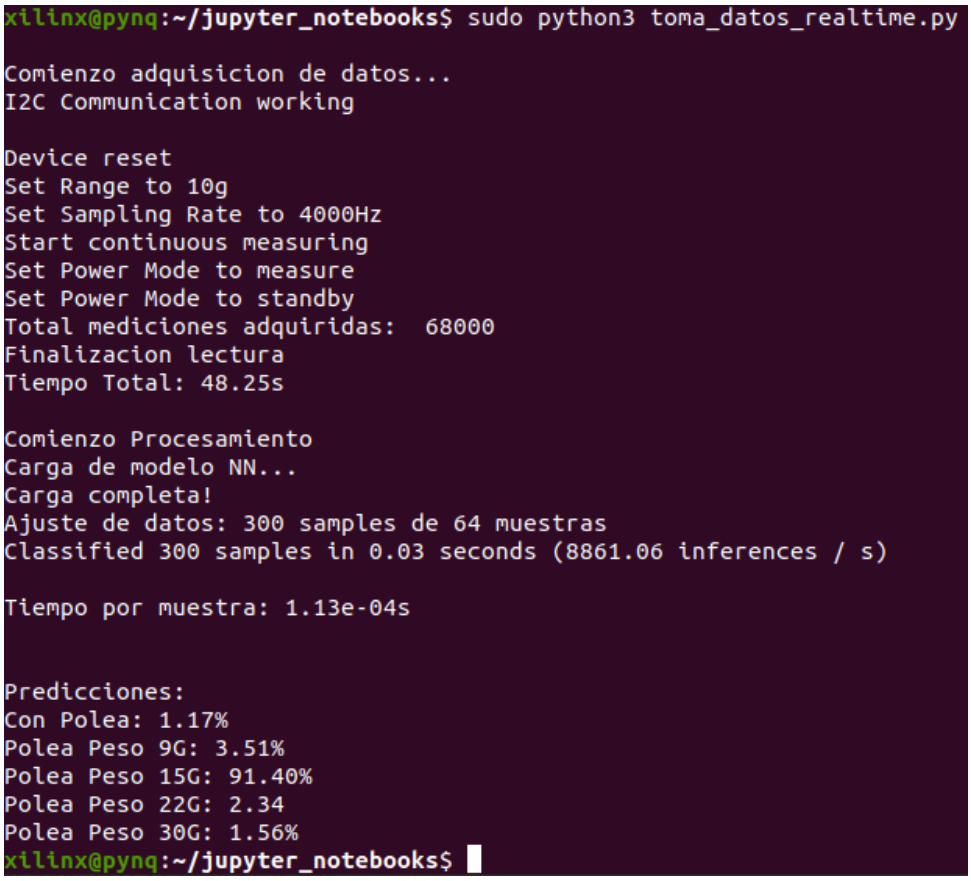


Figura 5.7: Resultados script real time en FGPA

Además, para completar y mejorar la validación del funcionamiento, se agregan al experimento algunos pesos distintos a los obtenidos en la primera adquisición de datos a fin de evaluar la predicción de la red frente a distintos escenarios a los cuales fue entrenada. Recordemos que los pesos son medidos con una balanza +/-1g y por lo tanto algún peso puede tener margen de error. Se toma como máximo peso la condición de funcionamiento Polea +30g por protección.

Condición	**Peso (g)**	**Predicción (%)**	**Condición Estimada**
Polea Sola	90	94.21	con_polea
Polea +7g	97	89.30	polea_peso_9g
Polea +9g	99	92.17	polea_peso_9g
Polea +13g	103	90.12	polea_peso_15g
Polea +15g	105	91.40	polea_peso_15g
Polea +22g	112	84.87	polea_peso_22g
Polea +24g	114	82.47	polea_peso_22g
Polea +30g	120	92.44	polea_peso_30g

Tabla 5.3: Resultados del experimento de polea con pesos en Zybo-Z7

Podemos observar en la Tabla 5.3 que la predicción de la red se comporta según lo esperado, pudiendo detectar las condiciones de funcionamiento iniciales y notando una pequeña caída para

pesos para los cuales no fue entrenada, mostrando que es capaz de identificar variaciones en las condiciones de funcionamiento, pero a medida que la condición de funcionamiento actual se aleja de la condición de funcionamiento inicial, la red empieza a perder presición en la predicción. Esto podría mejorarse buscando otras topologías de red más abarcativas o utilizando features distintos de entrada en vez de la señal directa sin procesar o también investigando técnicas de data augmentation para señales de vibración.

6 Conclusiones

Partiendo del análisis de los capítulos anteriores se pueden extraer un conjunto de conclusiones que permiten analizar el desarrollo realizado en este proyecto. Estas serán presentadas en este capítulo, junto con una sección donde se evaluarán posibles mejoras en el desarrollo del proyecto.

Por un lado se aprecia que el proceso de diseño de una red neuronal basada en sparse Autoencoder es un proceso complejo atado a muchas variables que pueden influir en el desarrollo y resultado del mismo. Como se había planteado, existen muchas topologías de red que pueden ser utilizadas para el análisis de vibraciones en el marco del mantenimiento predictivo y la elección de la topología queda a criterio de la persona que está diseñando la solución así como del set de datos que se va a utilizar. La cantidad de capas y nodos ocultos, el tamaño de input de la red, los tiempos de ejecución y entrenamiento, los parámetros de sparsity y el número de iteraciones de entrenamiento son algunas de las variables que hay que tomar en cuenta a la hora de diseñar la red.

En este proyecto se tomó como base el diseño de una red en particular para aplicar a un set de datos conocido. Se realizaron algunas adaptaciones y luego se realizó una validación experimental en la cual se obtuvieron mejores resultados a los obtenidos en el paper original. La precisión media del experimento fue de 96.09 % en comparación del 94.05 % de la investigación de referencia [12]. Es importante aclarar que este porcentaje se logró con una red mucho mas chica (64-40-5) e implementada en una FPGA, a diferencia de la original (512-300-150-5) que fue solo implementada en PC.

Por otro lado, también podemos destacar la validación del procedimiento y la investigación realizada en el último experimento donde se logró una precisión media del 94.82 % basándonos en un nuevo set de datos por las nuevas condiciones de funcionamiento, manteniendo la arquitectura de la red.

Se pudo comprobar que la red tuvo que ser adaptada y reducida para que funcionara con el set de datos experimental y que varios parámetros de los planteados en la investigación original no se pudieron reproducir o al aplicarlos de la misma manera no presentaron buenos resultados, por lo que podemos concluir que las conclusiones obtenidas en ese trabajo eran específicas para el experimento realizado. Al tratar de replicar los mismos parámetros no se obtuvieron resultados similares por lo que hubo que adaptar e investigar una mejor solución para el problema.

También es importante destacar la influencia del hardware utilizado en el proceso de este proyecto. Se utilizó una placa de bajos recursos Zybo Z7-10 la cual tiene una capacidad de procesamiento que queda limitada para esta aplicación en particular. Como se observó en la Figura 4.6, la elección del hardware tiene un gran impacto en los recursos disponibles y por consiguiente en las posibles limitaciones que se pueden presentar en el diseño de la solución necesaria. El uso de una placa de mayores recursos no solo permitiría el uso de una red más grande y posiblemente precisa sino que dejaría lugar a una mayor optimización en el procesamiento permitiendo también una mejora en los tiempos de ejecución.

Finalmente podemos concluir que la implementación de redes neuronales para el procesamiento de vibraciones para mantenimiento predictivo **es posible** incluso con placas de bajos recursos como la Zybo Z7-10. Se observó un marco de trabajo que permite una gran flexibilidad a la hora de su desarrollo y su implementación utilizando herramientas de código libre como

hls4ml. Este marco de trabajo permitió trabajar sobre varios escenarios de trabajo demostrando su flexibilidad por lo que podría trasladarse a escenarios más complejos con buena escalabilidad en placas de mayores recursos como nombramos. Este formato para armar prototipos de manera rápida no solo ofrece **muy buenos resultados** si se realiza un buen análisis de base, sino que luego permite otro tipo de implementaciones y optimizaciones ya conociendo la topología de red que mejor se adapta a la solución en caso de que hubiera condiciones que así lo requieran.

6.1. Futuras Mejoras

Esta sección explicará algunas limitaciones del diseño y como pueden ser solucionadas en futuros desarrollos. En principio, el diseño fue desarrollado para trabajar con un set de datos reconocido el cual se generó en un banco de ensayos profesional y de condiciones controladas. Sin embargo, se puede observar que el set de datos que se utiliza en este proyecto es más reducido y de menor calidad debido a una limitación en los costos de hardware. El uso de un banco de ensayos de condiciones controladas permite que la red sea más precisa y que se pueda obtener una mejor precisión en el procesamiento de vibraciones, así como una mayor confiabilidad en la diferenciación de las distintas condiciones de funcionamiento.

La disponibilidad de una FPGA de recursos limitados también jugo un papel importante en el desarrollo del proyecto. La capacidad de procesamiento de la placa es limitada y por lo tanto la red tendrá que ser adaptada para que funcione con el set de datos que se utiliza. El uso de un modelo con mayores recursos permitiría una mayor optimización en el procesamiento de la red y una mayor precisión en el procesamiento de vibraciones. El uso de una FPGA de mayores recursos también permitiría el análisis de otras topologías de redes neuronales que han demostrado mayor capacidad de procesamiento y análisis de vibraciones las cuales no eran posibles adaptarse a las limitaciones de hardware de la placa.

Otra de las mejoras posibles para este proyecto sería la de implementar la red neuronal elegida e implementarla en HDL sin utilizar hls4ml o HLS para su diseño. Si bien se pudo demostrar que la utilización de HLS es un método efectivo para la implementación de redes neuronales, el uso de HDL para su diseño permite una mejor utilización de los recursos de la placa y una mayor optimización en el procesamiento de la red (a un costo de mayor tiempo de desarrollo) lo que permitiría en principio poder implementar una red más grande sin utilizar más recursos en la FPGA.

Finalmente, uno de las mayores modificaciones al proyecto sería el cambio de topología de la red neuronal. El uso de otra topología podría mejorar considerablemente la precisión en el procesamiento de vibraciones y la confiabilidad de las pruebas, pero este cambio estaría condicionado por los recursos de la placa. Una optimización a nivel de programación de hardware permitirá una implementación de una red más grande con los mismos recursos, pero requeriría un trabajo mayor en su implementación limitando además la flexibilidad de la red para adaptarse a distintos escenarios. Este cambio sería útil cuando ya se tenga una topología definida así como sus características y ofrecería una rápida ejecución y procesamiento a coste de un mayor tiempo de desarrollo así como de conocimientos de arquitectura.

Bibliografía

[1] Vitis, "Vitis Unified Software Platform Documentation Application Acceleration Development," 1393. [Online]. Available: www.xilinx.com

[2] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," Tech. Rep., 2018. [Online]. Available: http://creativecommons.org/licenses/by/3.0/.

[3] S. HAYKIN, "Neural Networks: A Guided Tour," *Soft Computing and Intelligent Systems*, pp. 71–80, jan 2000.

[4] T. Evgeniou and M. Pontil, "Support vector machines: Theory and applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2049 LNAI, pp. 249–257, 2001.

[5] P. C. Fife, "Mathematical Aspects of Reacting and Diffusing Systems," vol. 28, 1979. [Online]. Available: http://link.springer.com/10.1007/978-3-642-93111-6

[6] D. P. Kroese, Z. I. Botev, T. Taimre, and R. Vaisman, "Data science and machine learning : mathematical and statistical methods," p. 513.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature 1986 323:6088*, vol. 323, no. 6088, pp. 533–536, 1986. [Online]. Available: https://www.nature.com/articles/323533a0

[8] Z.-L. Yang and X.-H. Wang, "Fast inference of deep neural networks in FPGAs for particle physics You may also like Implementation and verification of different ECC mitigation designs for BRAMs in flash-based FPGAs." [Online]. Available: https://doi.org/10.1088/1748-0221/13/07/P07027

[9] K. Guo, S. Zeng, J. Yu, Y. U. Wang, H. Yang, and Y. Wang, "A Survey of FPGA-Based Neural Network Inference Accelerator," Tech. Rep. 11, 2017.

[10] Z.-L. Yang, X.-H. Wang, A. , L. Wang, J. Lai, J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in FPGAs for particle physics You may also like Implementation and verification of different ECC mitigation designs for BRAMs in flash-based FPGAs A new FPGA architecture suitable for DSP applications Fast inference ," *PPPPPPPPP*, vol. 10, 2018. [Online]. Available: https://doi.org/10.1088/1748-0221/13/07/P07027

[11] "Xilinx/PYNQ: Python Productivity for ZYNQ." [Online]. Available: https://github.com/Xilinx/PYNQ

[12] H. Shao, H. Jiang, H. Zhao, and F. Wang, "A novel deep autoencoder feature learning method for rotating machinery fault diagnosis," *Mechanical Systems and Signal Processing*, vol. 95, pp. 187–204, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.ymssp.2017.03.034

[13] T. Amaral, L. M. Silva, L. A. Alexandre, C. Kandaswamy, J. M. Santos, and J. M. De Sá, "Using different cost functions to train stacked auto-encoders," *Proceedings - 2013 12th Mexican International Conference on Artificial Intelligence, MICAI 2013*, pp. 114–120, 2013.

[14] J. Zabalza, J. Ren, J. Zheng, H. Zhao, C. Qing, Z. Yang, P. Du, and S. Marshall, "Novel segmented stacked autoencoder for effective dimensionality reduction and feature extraction in hyperspectral imaging," *Neurocomputing*, vol. 185, pp. 1–10, apr 2016.

[15] W. Ma, H. Qu, G. Gui, L. Xu, J. Zhao, and B. Chen, "Maximum correntropy criterion based sparse adaptive filtering algorithms for robust channel estimation under non-Gaussian environments," *Journal of the Franklin Institute*, vol. 352, no. 7, pp. 2708–2727, jul 2015.

[16] "Bearing Data Center | Case School of Engineering | Case Western Reserve University." [Online]. Available: https://engineering.case.edu/bearingdatacenter

[17] S. Zhang, S. Zhang, B. Wang, and T. G. Habetler, "Deep Learning Algorithms for Bearing Fault Diagnosticsx - A Comprehensive Review," *IEEE Access*, vol. 8, pp. 29 857–29 881, 2020.

[18] D. Neupane and J. Seok, "Bearing fault detection and diagnosis using case western reserve university dataset with deep learning approaches: A review," *IEEE Access*, vol. 8, pp. 93 155–93 178, 2020.

[19] M. Sohaib and J. M. Kim, "Reliable Fault Diagnosis of Rotary Machine Bearings Using a Stacked Sparse Autoencoder-Based Deep Neural Network," *Shock and Vibration*, vol. 2018, 2018.

[20] A. Ng, "CS294A Lecture notes Sparse autoencoder."

[21] Q. He, X. Hu, H. Ren, and H. Zhang, "A novel artificial fish swarm algorithm for solving large-scale reliability–redundancy application problem," *ISA Transactions*, vol. 59, pp. 105–113, nov 2015.

[22] W. Liu, P. P. Pokharel, and J. C. Principe, "Correntropy: Properties and Applications in Non-Gaussian Signal Processing," *IEEE TRANSACTIONS ON SIGNAL PROCESSING*, vol. 55, no. 11, 2007.

7 Apéndice

7.1. Estimación Inicial de Tareas

En esta sección se presenta la estimación inicial de tareas necesarias para la realización de este proyecto.

- Selección de arquitectura de red neuronal
  - Investigación de red a utilizar a partir de otras investigaciones (5 semanas)
  - Implementación en PC utilizando datasets (4 semanas)
  - Pruebas de confiabilidad (4 semanas)
- Implementación de arquitectura en C
  - Investigación de sensor a utilizar (2 semanas)
  - Lectura de datos en placa de prueba (2 semanas)
  - Desarrollo de driver en C (2 semanas)
- Armado de Kit de pruebas
  - Investigación de kit de prueba (3 semanas)
  - Montaje del kit (1 semana)
  - Diseño de montaje del sensor (2 semanas)
- Implementación de arquitectura en FPGA
  - Investigación de arquitectura de FPGA (5 semanas)
  - Obtención de datos a través de protocolo (2 semanas)
  - Implementación de la red neuronal elegida (6 semanas)
  - Pruebas con los datos de entrenamiento (2 semanas)
  - Pruebas con kit de pruebas (3 semanas)
- Pruebas de confiabilidad (5 semanas)
  - Obtención de conclusiones (1 semana)
  - Informe Parcial (2 semanas)
  - Informe Final (4 semanas)

7.2. Gestión de Riesgos

En esta sección se presentan los riesgos iniciales planteados para este proyecto.

- Recursos insuficientes de FPGA: **Riesgo Medio/Bajo**
  - Es un riesgo del proyecto, el hardware es limitado a la placa provista por el Departamento de Electrónica y debe implementarse ahí. El riesgo existe, pero es

poco probable, ya que se han implementado redes de procesamiento de imagen que deberían requerir más recursos que la probable red a implementar.
    - Para mitigar el riesgo, se puede buscar alguna alternativa de arquitectura de red más liviana aunque signifique perder exactitud.
    - En caso de que no se pueda realizar ninguna implementación con redes neuronales, se buscará realizar un análisis clásico de la medición para realizar algún tipo de predicción.
- Imposibilidad de implementar la red elegida en la FPGA: **Riesgo Alto**
    - Dado que se trata de una implementación en hardware existe la posibilidad que al intentar bajar la implementación a la placa surjan problemas que hagan que la red elegida no se pueda implementar o no tenga una respuesta aceptable para los parámetros propuestos.
    - También existe la posibilidad de que por falta de conocimientos/recursos no se pueda implementar de manera correcta en tiempo y forma.
    - Como alternativa para mitigar la imposibilidad de la implementación se puede implementar la red en una arquitectura soportada por ejemplo en la placa Beaglebone Black para probar la funcionalidad.
- Incorrecta selección de arquitectura a implementar: **Riesgo Medio**
    - El riesgo implica una necesidad de volver a implementar que consumirá tiempo al tener la necesidad de modificar la red investigada.
    - Se puede mitigar mejorando el proceso de investigación y pruebas antes de realizar la implementación final.
- Imposibilidad de realizar mediciones físicas: **Riesgo Alto**
    - Partiendo de la imposibilidad de acceder a equipamiento físico más apto para el proyecto debido a la cuarentena, se armará un banco de ensayos personal con elementos intentando simular los objetivos del proyecto, pero existe el riesgo de que no logren reproducir un escenario ideal y haya problemas en la implementación y lectura de mediciones con el sensor elegido.
    - Como alternativa, se podrá probar la implementación de la red elegida utilizando una entrada de datos de prueba simulando distintos equipamientos para demostrar que la red funciona correctamente y que el problema de la implementación está en la toma de muestras y no en la investigación o implementación del proyecto.
- Mal manejo o estimación de tiempos: **Riesgo Alto**
    - Pueden existir tareas que hayan sido mal estimadas o tareas que no se consideran o se priorizaron mal. Esto provocaría que el cronograma propuesto no se cumpla y no se llegue con los tiempos estimados en las fechas propuestas.
    - Como alternativa se intentará ir logrando hitos acumulativos que permitan llegar con un producto mínimo viable (MVP) para mostrar la funcionalidad aunque sea reducida.
- Riesgos externos: **Riesgo Alto**
    - Debido a la situación actual de público conocimiento puede haber factores externos

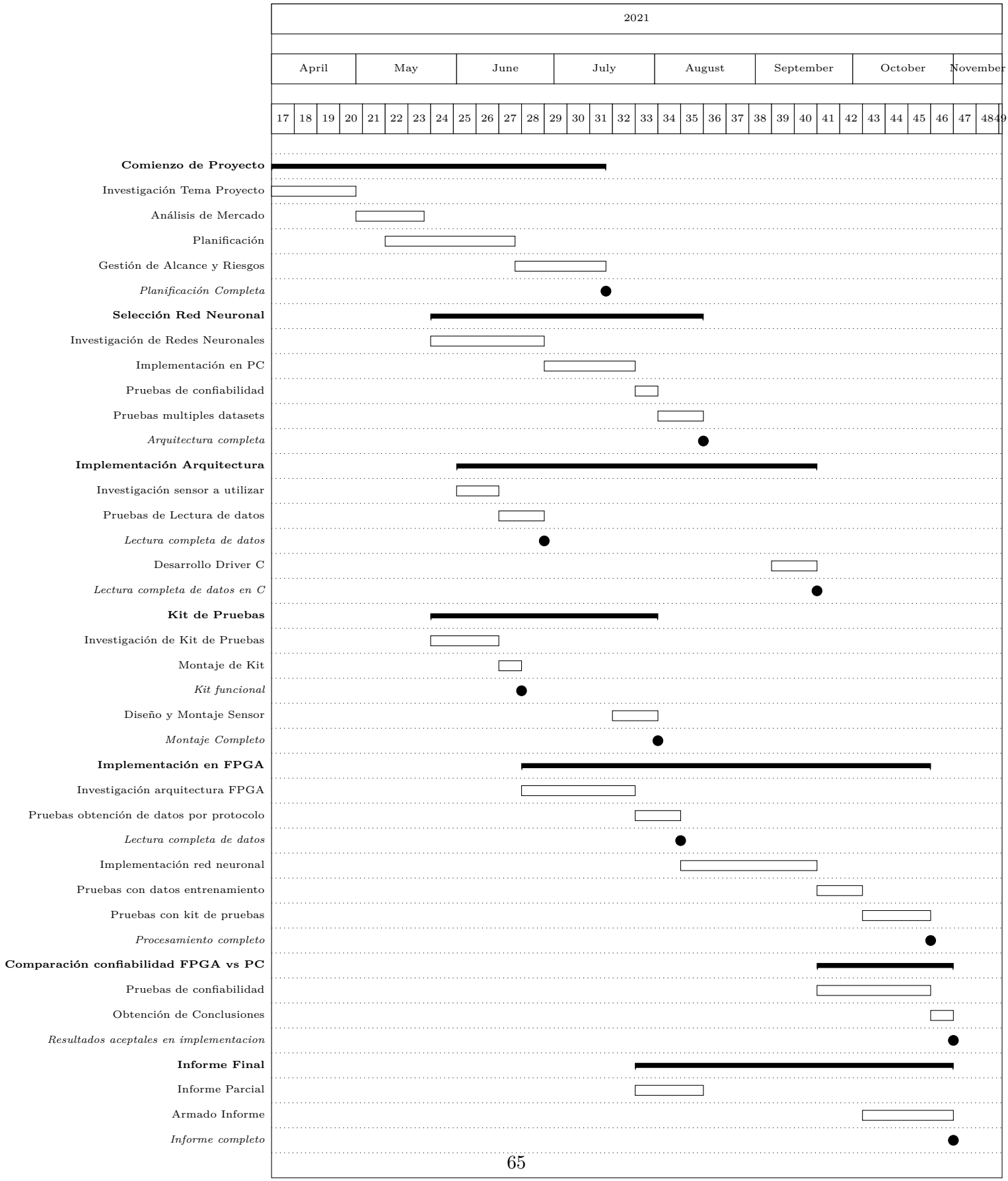
relacionados con el Covid-19 que impacten de lleno en el correcto desarrollo del proyecto.

7.3. Gestión de Tiempo

Este apéndice presenta el Diagrama de Gantt propuesto al iniciar el proyecto, con la descripción de cada una de las tareas.

- **Comienzo del Proyecto**
  - **Investigación tema proyecto**: Esta tarea contempla la investigación inicial sobre el tema a tratar en el desarrollo del proyecto.
  - **Análisis de mercado**: Esta tarea contempla el análisis de mercado sobre productos similares al propuesto en el desarrollo del proyecto.
  - **Planificación**: Esta tarea contempla la planificación del desarrollo del proyecto y la definición de los objetivos del mismo.
  - **Gestión de Alcance y Riesgos**: Esta tarea contempla la gestión de los alcances y riesgos del proyecto.
- **Selección Red Neuronal**
  - **Investigación de Redes Neuronales**: Esta tarea contempla la investigación de diferentes arquitecturas de redes neuronales para el proyecto.
  - **Implementación en PC**: Esta tarea contempla la implementación de la red neuronal en PC a fin de verificar su funcionamiento.
  - **Pruebas de Confiabilidad**: Esta tarea contempla la prueba de confiabilidad de la red neuronal basándonos en un set de datos conocido.
- **Implementación Arquitectura**
  - **Investigación sensor a utilizar**: Esta tarea contempla la investigación de diferentes sensores de vibraciones para el proyecto.
  - **Prueba lectura de datos**: Esta tarea contempla la prueba de lectura de datos con el sensor elegido.
  - **Desarrollo de driver**: Esta tarea contempla el desarrollo de un driver para la adquisición de datos con el sensor elegido.
- **Kit de Pruebas**
  - **Investigación de Kit de Pruebas**: Esta tarea contempla la investigación de diferentes kits de pruebas y elementos para el proyecto.
  - **Montaje de Kit**: Esta tarea contempla el montaje de los elementos del kit de pruebas.
  - **Diseño y Montaje del Sensor**: Esta tarea contempla el armado y pruebas del sensor a utilizar.
- **Implementación en FPGA**
  - **Investigación arquitectura FPGA**: Esta tarea contempla la investigación de diferentes arquitecturas de FPGA para el proyecto.

    - **Pruebas obtención de datos por protocolo**: Esta tarea contempla la prueba de obtención de datos por protocolo a través de la FPGA utilizada.
    - **Implementación red neuronal**: Esta tarea contempla la implementación de la red neuronal en la FPGA.
    - **Pruebas con datos entrenamiento**: Esta tarea contempla la prueba de la red neuronal con datos de entrenamiento.
    - **Pruebas con kit de pruebas**: Esta tarea contempla la prueba de la red neuronal con datos de prueba obtenidos con el banco de pruebas.
- **Comparación confiabilidad FPGA vs PC**
    - **Pruebas de confiabilidad**: Esta tarea contempla las pruebas finales de la red neuronal basándonos en el set de datos propio.
    - **Obtención de Conclusiones**: Esta tarea contempla la obtención de conclusiones sobre la confiabilidad de la red neuronal y su aplicación para el proyecto.
- **Informe Final**: Esta tarea contempla el armado del informe final del proyecto.


2021
April
May
June
July
August
September
October
November
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
Comienzo de Proyecto
Investigación Tema Proyecto
Análisis de Mercado
Planificación
Gestión de Alcance y Riesgos
Planificación Completa
Selección Red Neuronal
Investigación de Redes Neuronales
Implementación en PC
Pruebas de confiabilidad
Pruebas multiples datasets
Arquitectura completa
Implementación Arquitectura
Investigación sensor a utilizar
Pruebas de Lectura de datos
Lectura completa de datos
Desarrollo Driver C
Lectura completa de datos en C
Kit de Pruebas
Investigación de Kit de Pruebas
Montaje de Kit
Kit funcional
Diseño y Montaje Sensor
Montaje Completo
Implementación en FPGA
Investigación arquitectura FPGA
Pruebas obtención de datos por protocolo
Lectura completa de datos
Implementación red neuronal
Pruebas con datos entrenamiento
Pruebas con kit de pruebas
Procesamiento completo
Comparación confiabilidad FPGA vs PC
Pruebas de confiabilidad
Obtención de Conclusiones
Resultados aceptales en implementacion
Informe Final
Informe Parcial
Armado Informe
Informe completo