Universidad Zaragoza

Trabajo Fin de Máster

Implementación de un acelerador para redes neuronales bayesianas en plataformas RISC-V

Implementation of an Acceleration Unit for Bayesian Neural Networks in RISC-V Platforms

Autor

Samuel Pérez Pedrajas

Directores

Javier Resano Ezcaray

Darío Suárez Gracia

Titulación

Máster Universitario en Ingeniería Informática

ESCUELA DE INGENIERÍA Y ARQUITECTURA

2024

AGRADECIMIENTOS

Este trabajo ha sido parcialmente financiado por el Programa de Becas y Ayudas del Instituto de Investigación en Ingeniería de Aragón (I3A).

RESUMEN

Las redes neuronales son modelos de aprendizaje automático muy populares para tareas de clasificación. La combinación del internet de las cosas y el aprendizaje automático es un paradigma que también está creciendo en popularidad y cada vez más redes neuronales se están ejecutando en dispositivos de bajo consumo y prestaciones. La fiabilidad de las predicciones de los modelos de aprendizaje automático es algo crucial en aplicaciones relacionadas con la seguridad y las redes neuronales tradicionales no son capaces de proveer esta fiabilidad.

Las redes neuronales bayesianas mejoran su fiabilidad calculando métricas de incertidumbre asociadas a sus predicciones. Sin embargo, generar estas métricas aumenta sus requisitos computacionales. Este trabajo busca soluciones a los desafíos de ejecutar este tipo de redes en dispositivos de bajas prestaciones.

En primer lugar, se ha desarrollado una biblioteca para la inferencia de redes neuronales bayesianas en dispositivos de bajas prestaciones.

En segundo lugar, se han desarrollado técnicas software para optimizar la ejecución. Estas redes requieren ejecutar múltiples propagaciones de la entrada en las que los pesos se muestrean a partir de distribuciones estadísticas, normalmente Gaussianas. Este proceso de muestreo puede llegar a ocupar el 85.13 % del tiempo de ejecución. Por ello, se han desarrollado distintas técnicas para optimizar este muestreo, integrando estas optimizaciones en la biblioteca desarrollada, obteniendo tiempos de ejecución en promedio 4.57 veces menores.

Finalmente, en este trabajo se ha desarrollado una extensión del conjunto de instrucciones RISC-V de bajo coste para acelerar la inferencia de redes neuronales bayesianas. Esta extensión se ha implementado en un procesador y se ha estudiado su coste energético y hardware en una FPGA. Esta extensión añade instrucciones nuevas que optimizan el muestreo de pesos mediante un generador de números pseudoaleatorios Gaussianos hardware, obteniendo tiempos de ejecución hasta 7.65 veces menores provocando un ahorro energético en un factor similar.

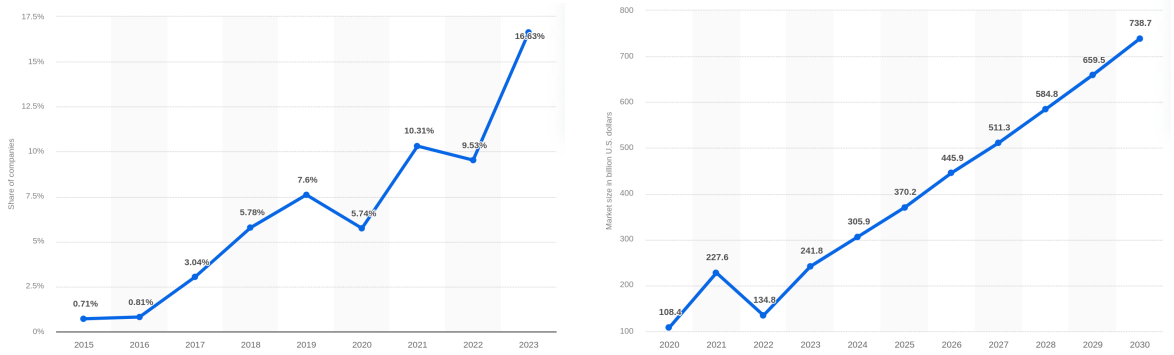
Índice

Capítulo 1

Introducción

1.1. Motivación

La inteligencia artificial (IA) gana cada año más popularidad, impulsada por los últimos avances en tecnología y su aplicación en diversas industrias. La Figura 1.1a muestra el porcentaje de empresas a nivel mundial que muestran interés en la IA y en la Figura 1.1b se representa el crecimiento del valor del mercado de la IA en miles de millones de dólares.



(a) Interés de las empresas, representado mediante el porcentaje de empresas [1].

(b) Valor de mercado en miles de millones de dólares [2].

Figura 1.1: Crecimiento de la IA en los últimos 10 años y crecimiento esperado hasta 2030.

Al aumentar este interés y por tanto su uso en diversos campos también lo hará el consumo energético. La motivación principal de este trabajo es mejorar el rendimiento y la eficiencia energética de la inferencia en un subconjunto de técnicas de aprendizaje automático. Concretamente, este trabajo se centra en las redes neuronales bayesianas (**B***ayesian* **N***eural* **N***etwork*). Estas redes calculan la incertidumbre de sus predicciones aportando transparencia en su uso. En relación, otra de las motivaciones de este trabajo es facilitar la adopción de estas redes para ayudar hacia el objetivo de la Unión Europea de trabajar con IAs más transparentes [3]. Otra de las motivaciones principales de este trabajo es fomentar el desarrollo de la arquitectura RISC-V, que es una arquitectura de procesadores abierta y libre. Europa ha apostado por esta arquitectura, mediante el proyecto *European*

Processor Initiative [4], para que sus empresas puedan diseñar y producir sus propios procesadores sin las restricciones y costos asociados con las arquitecturas propietarias tradicionales.

Específicamente, este TFM aborda la ejecución de la inferencia de BNNs en dispositivos de bajas prestaciones orientados hacia el internet de las cosas (**I***nternet* **o***f* **T***hings*). Esta combinación se conoce comúnmente como TinyML y tiene un gran potencial a la hora de reducir el consumo energético [5]. Los componentes IoT son más eficientes energéticamente que los procesadores de altas prestaciones, por lo que trasladar la ejecución de la inferencia de BNN a estos componentes reduciría su consumo además de eliminar comunicaciones costosas entre componentes IoT desplegados y servidores.

1.2. Objetivos y alcance

El objetivo principal de este trabajo es optimizar la inferencia de BNN en un procesador RISC-V de bajas prestaciones. Se han seguido los siguientes pasos para lograr el objetivo:

1. Estudiar el funcionamiento de la inferencia de las BNN.
2. Analizar aceleradores existentes de inferencia de BNN.
3. Investigar el soporte existente para ejecutar inferencia de BNN en procesadores de bajas prestaciones sin precisión de punto flotante.
4. Implementar una biblioteca en C junto a herramientas que permitan ejecutar la inferencia de BNN en un procesador RISC-V desarrollado en un trabajo previo [6].
5. Validar la biblioteca implementada con diferentes modelos.
6. Analizar la carga de trabajo de la biblioteca desarrollada.
7. Desarrollar optimizaciones software y analizar sus ventajas y desventajas.
8. Investigar sobre generadores de números aleatorios gaussianos (**G***aussian* **R***andom* **N***umber* **G***enerator*).
9. Diseñar un GRNG que se pueda integrar en el procesador RISC-V como una extensión.
10. Estudiar los pasos necesarios para extender un procesador RISC-V y poder utilizar la extensión desde código en alto nivel C.
11. Analizar el coste y rendimiento de la extensión utilizando una *Field Programmable Gate Array* (FPGA).

1.3. Diseminación de resultados

Los resultados de este trabajo han sido presentados en la conferencia IEEE NANO 2024. También han sido presentados en la Jornada de Jóvenes Investigadores 2024 del I3A como presentación oral.

Todo el código de este trabajo se encuentra disponible en un repositorio público en GitHub [7].

1.4. Estructura del documento

El Capítulo 2 describe el estado del arte relacionado con la arquitectura RISC-V y las BNN. El Capítulo 3 describe la metodología que se ha seguido durante el desarrollo del trabajo y las herramientas y modelos utilizados. En este capítulo también se explica la configuración experimental utilizada y los componentes que se han desarrollado para crearla. El Capítulo 4 explica las decisiones de diseño y desarrollo de una biblioteca para ejecutar inferencia de BNN en plataformas RISC-V y muestra resultados obtenidos con la misma. El Capítulo 5 expone dos optimizaciones software para la inferencia de BNN implementadas en la biblioteca mencionada previamente y los resultados que se han obtenido con ambas. El Capítulo 6 explica cómo se ha implementado una extensión para la inferencia de BNN en un procesador RISC-V y los resultados que se han obtenido con ella así como su eficiencia energética. Por último el Capítulo 7 contiene las conclusiones y posible trabajo futuro.

Capítulo 2

Estado del arte

2.1. RISC-V

RISC-V es una arquitectura libre de conjunto de instrucciones reducido (**R***educed* **I***nstruction* **S***et* **C***omputer*) [8]. A diferencia de las arquitecturas propietarias tradicionales, RISC-V ofrece una especificación libre, lo que permite diseñar nuevos sistemas hardware sin tener que licenciar la arquitectura.

RISC-V se caracteriza por su simplicidad y modularidad, esto hace que sea una buena solución para diseñar tanto procesadores de muy altas prestaciones como procesadores simples para dispositivos empotrados. El tamaño del repertorio mínimo de instrucciones de 32 bits es de 40 instrucciones. Las extensiones se dividen en privilegiadas y no privilegiadas, dependiendo del modo de ejecución para el que estén enfocadas. Los modos privilegiados otorgan más control sobre el procesador y existen para dar soporte a programas como sistemas operativos e hipervisores. La mayoría de programas se ejecutan en modo no privilegiado. Sobre este repertorio hay 43 extensiones no privilegiadas, que dan soporte por ejemplo a la coma flotante, operaciones atómicas, ... y 16 extensiones privilegiadas, como por ejemplo la extensión de virtualización, recogidas en el manual de la arquitectura [9, 10].

La filosofía de diseño RISC-V consiste en partir de un conjunto de instrucciones base sencillo y extenderlo con instrucciones de dominio específico para obtener un diseño final lo más óptimo posible pero que mantenga la flexibilidad que puede ofrecer un procesador de propósito general. Una desventaja de esta filosofía es que al adaptar un programa para utilizar diferentes extensiones su portabilidad a otros chips se reduce.

Siguiendo la filosofía de diseño RISC-V, en la actualidad, empresas como Codasip [11] o Synopsis [12] ofrecen diseños de procesadores base junto con entornos de desarrollo para extenderlos mediante instrucciones específicas y crear procesadores a medida para sus clientes. También existen entornos y procesadores abiertos, como Sargantana [13], PULP [14] y Rocket [15].

Este trabajo utiliza un procesador RISC-V sencillo de 32 bits segmentado en 5 etapas con ejecución en orden que implementa las instrucciones estándar RV32IM [6]. RV32I indica que soporta el conjunto de instrucciones base para enteros de 32 bits. M indica soporta el modo de ejecución *Machine*, lo que implica el soporte a interrupciones, excepciones y a la extensión Zicsr, que permite interactuar con registros de control. La Figura 2.1 muestra un esquema simplificado de la ruta de datos del procesador. Este procesador es un ejemplo perfecto de un procesador de bajo coste y prestaciones que puede encontrarse en un sistema empotrado.

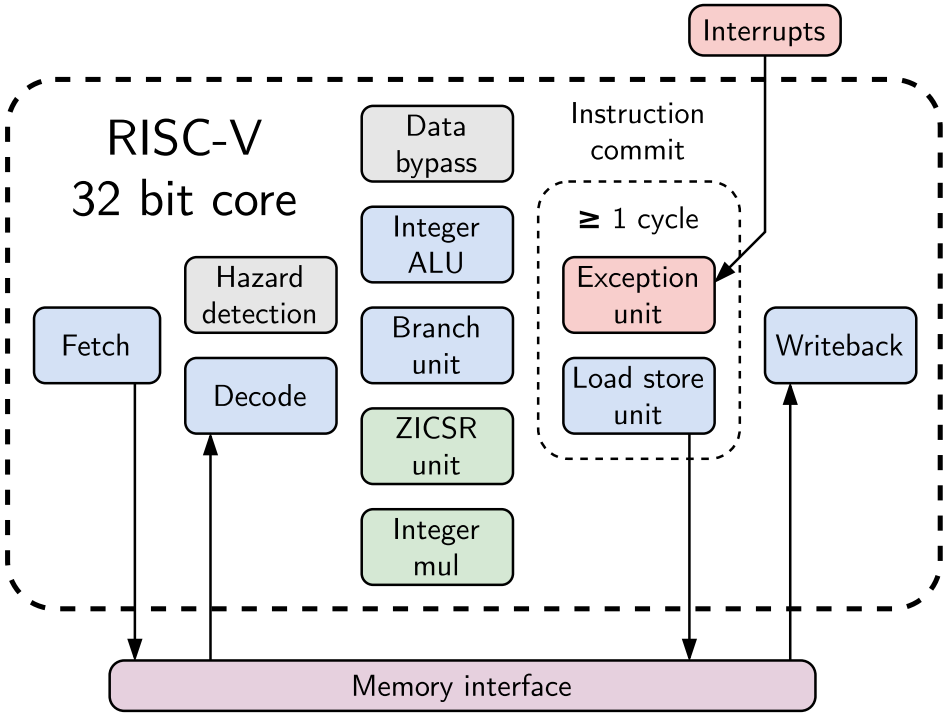


Figura 2.1: Ruta de datos del procesador RISC-V utilizado. Los bloques relacionados con la arquitectura base RV32I se muestran en azul, los bloques para detectar riesgos de datos en gris, los bloques relacionados con el modo M en rojo, los bloques de extensiones estándar extra en verde y el interfaz con el bus de memoria en morado.

2.2. Redes neuronales bayesianas

El aprendizaje automático es una rama de la inteligencia artificial que se centra en el desarrollo de algoritmos y modelos estadísticos que permiten generalizar respuestas en problemas para los que no han sido explícitamente programados aprendiendo de un conjunto de datos dado.

Las redes neuronales (**N***eural* **N***etworks*) son modelos de aprendizaje automático cuya estructura está inspirada en el funcionamiento de redes de neuronas biológicas [16]. Una NN está compuesta por capas de neuronas conectadas entre sí. Cada capa cuenta con

un conjunto de pesos y una función de activación. Las NN necesitan un proceso de entrenamiento en el que son expuestas a un conjunto de datos para que ajusten el valor de los pesos de las diferentes capas. En los últimos años, las NN han obtenido muy buenos resultados en problemas de clasificación y regresión entre otros [17].

Las BNN son un tipo de NN que integran modelado probabilístico, lo que les permite cuantificar la incertidumbre en tareas de aprendizaje automático, mejorando su confianza y fiabilidad [18]. Sin embargo, necesitan más parámetros para definir los pesos, normalmente el doble que una NN convencional, y la inferencia es más compleja. Los pesos de las BNN son distribuciones de probabilidad, normalmente distribuciones Gaussianas, que se han de muestrear durante las propagaciones de la entrada de la red. El algoritmo de inferencia más común para las BNN requiere múltiples propagaciones, aumentando drásticamente el coste computacional del mismo [19].

La Figura 2.2 muestra la utilidad de las métricas de incertidumbre con un ejemplo simple. Considerando una NN convencional y una BNN ambas entrenadas para clasificar imágenes de perros y gatos, en el caso de que se les pidiera clasificar un dato anómalo, como la imagen de un tigre, con la BNN se obtendría un valor de incertidumbre alto, indicando una anomalía en la predicción, mientras que con una NN convencional no se tendría ningún indicio de que el dato de entrada era anómalo.

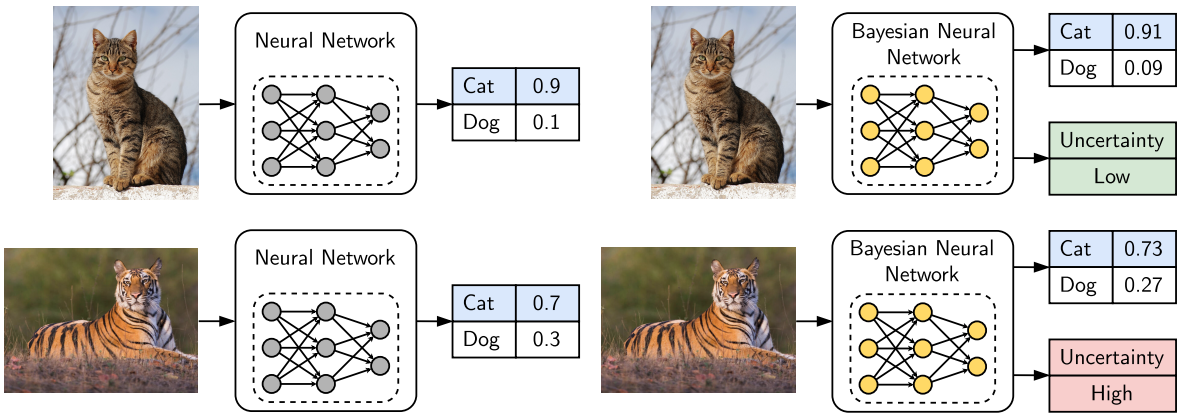
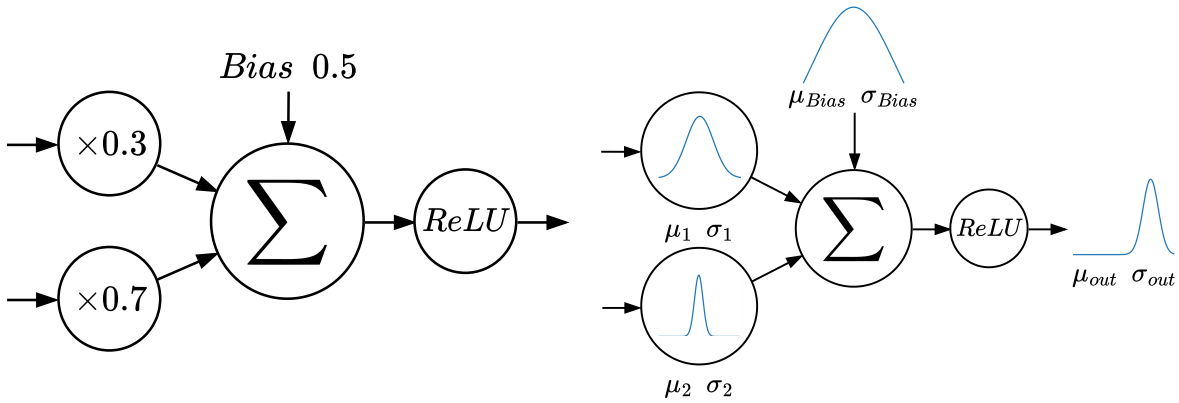


Figura 2.2: Ejemplo sencillo de utilidad de las métricas de incertidumbre aportadas por las BNN. Una NN convencional y una BNN, ambas entrenadas para clasificar imágenes de perros y gatos, reciben como entrada la imagen de un tigre. La BNN es capaz de detectar el dato anómalo mientras que la NN convencional no.

La Figura 2.3 muestra una comparación sencilla de una neurona clásica, Subfigura 2.3a, y de una neurona bayesiana, Subfigura 2.3b, ambas con una función de activación de rectificación (**Re***ctified* **L***inear* **U***nit*). En el caso de la neurona bayesiana los pesos son distribuciones Gaussianas con diferentes medias y desviaciones típicas.

En un entorno IoT donde los datos pueden ser incompletos o ruidosos la capacidad de cuantificar la incertidumbre es crucial para tomar decisiones confiables en aplicaciones



(a) Neurona clásica (b) Neurona bayesiana

Figura 2.3: Comparación de una neurona clásica con respecto a una neurona bayesiana, ambas con la misma función de activación ReLU. Los pesos de la neurona convencional son valores estáticos mientras que los de la neurona bayesiana son distribuciones gaussianas parametrizadas mediante medias y desviaciones típicas.

críticas. Por lo que las BNN son una muy buena herramienta para mejorar la precisión y la robustez de los sistemas de aprendizaje automático en estos entornos.

Las métricas de incertidumbre proporcionadas por las BNN son muy útiles para detectar diferentes tipos de situaciones. Por ejemplo, pueden servir para valorar la confianza de la predicción, si una predicción tiene una alta incertidumbre podría indicar que es menos confiable y que requeriría de verificación humana. También pueden utilizarse para valorar la calidad de los datos de entrenamiento, ya que si los datos de entrenamiento contienen muestras mal etiquetadas sus predicciones tendrán una mayor incertidumbre asociada. Alcolea *et al.* mostraron que existe una clara correlación entre la incertidumbre y la precisión en un modelo bien entrenado, por tanto la incertidumbre se puede utilizar para identificar salidas con menor probabilidad de acierto de la requerida [18]. También observaron que al entrenar una BNN con los datos de 2 clases mezclados se aprecia un claro incremento en la incertidumbre de las predicciones asociadas a esas clases, y que la incertidumbre aumentaba cuando se añadía ruido aleatorio a las entradas. Otro posible uso es detectar que los datos de entrenamiento no reflejan correctamente la realidad, predicciones con alta incertidumbre pueden indicar que los datos que está procesando el modelo no se parecen a los datos con los que ha sido entrenado. En entornos dinámicos como IoT, donde las condiciones pueden cambiar rápidamente, si la incertidumbre en las predicciones aumenta repentinamente, puede indicar un cambio en el entorno que requiera una recalibración del modelo.

2.2.1. Fundamentos teóricos

Estas redes neuronales se basan el Teorema de Bayes, mostrado en la Ecuación 2.1, para modelar la probabilidad de un conjunto de pesos w dado un conjunto de datos de entrenamiento $D = \{x, y\}$. Esta distribución de probabilidad es la probabilidad a posteriori.

$$p(w|D) = \frac{p(D|w)p(w)}{p(D)} \tag{2.1}$$

El entrenamiento de una BNN consiste en calcular esta distribución a posteriori. Calcular esta distribución para un modelo grande y complicado como una BNN es intratable. Para aproximar esta distribución se utilizan métodos de inferencia bayesiana. El método más común es la inferencia variacional, que aproxima la distribución a posteriori usando una más simple $q_\phi(w)$. Esta aproximación se realiza minimizando la divergencia de Kullback-Leibler entre $q_\phi(w)$ y $p(w|D)$ mediante la actualización de un conjunto de parámetros ϕ [20]. Teóricamente, las distribuciones podrían tener cualquier forma, pero para reducir el espacio de búsqueda sólo se utilizan distribuciones simétricas y simples. Las distribuciones Gaussianas son una buena opción porque solo se definen con dos parámetros, media y desviación típica, lo que ayuda a reducir el tamaño del modelo.

En consecuencia, una BNN entrenada consiste en un conjunto de medias y desviaciones de diferentes distribuciones Gaussianas $\phi = \{\mu, \sigma\}$, lo que significa que los pesos se convierten en distribuciones de probabilidad en lugar de valores fijos, y la salida también se convierte en una distribución en lugar de un valor único. Esto permite medir la incertidumbre de la predicción. La Ecuación 2.2 muestra la distribución de predicción a posteriori de una nueva observación x^*.

$$p(y^*|x^*, D) = \int p(y|x^*, w)p(w|D)dw \tag{2.2}$$

Mediante métodos de Monte Carlo se puede aproximar la integral de la Ecuación 2.2. En este caso, las muestras son diferentes propagaciones estocásticas de la entrada. En cada una de estas propagaciones, los pesos se muestrean de las distribuciones Gaussianas que forman la distribución a posteriori aproximada.

Siendo T el número de muestras, K el número de clases posibles y una muestra a_t el vector de longitud K de componentes $p(y^* = c_k|x^*, w_t)$. Cada uno de sus componentes representa la probabilidad de que y^* sea c_k. Para obtener una única predicción se puede tomar el componente máximo del vector promedio de todas las muestras.

Cálculo de la incertidumbre

En problemas de clasificación, se puede medir la incertidumbre de una predicción utilizando las diferentes muestras Monte Carlo obtenidas [21]. A continuación se explica como obtener las métricas más relevantes y lo que representan.

La incertidumbre predictiva $\mathbb{H}$ representa la incertidumbre de una predicción en el rango $[0, \log(K)]$ y puede calcularse utilizando la Ecuación 2.3.

$$\mathbb{H}(y|x, D) = -\sum_{k=1}^{K} \left[\left(\frac{1}{T} \sum_{t=1}^{T} a_t \right) \log \left(\frac{1}{T} \sum_{t=1}^{T} a_t \right) \right] \tag{2.3}$$

La incertidumbre puede dividirse en dos, aleatoria y epistémica, $\mathbb{H}$ captura ambas. La entropía esperada $\mathbb{E}p$ captura la incertidumbre aleatoria, que es causada por ambigüedades en el conjunto de datos como mediciones ruidosas, clases superpuestas o muestras mal etiquetadas. $\mathbb{E}p$ puede calcularse utilizando la Ecuación 2.4.

$$\mathbb{E}p(w|D)[\mathbb{H}(y|x, D)] = \frac{1}{T} \sum^{T} t = 1 \left(-\sum_{k=1}^{K} a_t \log(a_t) \right) \tag{2.4}$$

La incertidumbre epistémica representa lo que el modelo no sabe debido a la falta de datos durante el proceso de entrenamiento, y puede calcularse utilizando la información mutua (**M***utual* **I***nformation*) mostrada en la Ecuación 2.5.

$$MI(y, w|x, D) = \mathbb{H}(y|x, D) - \mathbb{E}_{p(w|D)}[\mathbb{H}(y|x, D)] \tag{2.5}$$

2.2.2. Aceleración hardware

La aceleración de algoritmos de aprendizaje automático es un campo de investigación muy activo con muchas propuestas recientes [22]. La aceleración de las BNN no es una excepción, debido en parte al elevado coste de su algoritmo de inferencia. Otros trabajos se han centrado en desarrollar aceleradores de alto rendimiento para el proceso completo de inferencia [23, 24, 25].

Awano *et al.* propusieron un acelerador sin múltiples propagaciones ni muestreo para el algoritmo de inferencia de las BNN, reemplazando la función de activación ReLU por una función cuadrática [24]. Su trabajo muestra buenos resultados para el conjunto de datos MNIST, pero otros trabajos han demostrado que este enfoque no es generalizable y no funciona bien con otros modelos [25]. Por esta razón, este trabajo se centra en el algoritmo de múltiples propagaciones con muestreo, ya que es el método más estándar y ha sido ampliamente demostrado que da buenos resultados para diferentes conjuntos de datos [23, 25, 18].

El muestreo de distribuciones para generar los pesos es una parte fundamental de los trabajos que siguen el enfoque de múltiples propagaciones. Cai *et al.* propusieron un acelerador de inferencia de BNN con 2 posibles GRNG [23]. Uno basado en el teorema central del límite (TCL) utilizando la distribución binomial y otro en el método de Wallace [26]. Ambos obteniendo buenos resultados de precisión y aceleración.

Los GRNG son un tema que ya ha sido explorado en profundidad [27]. Trabajos anteriores han mostrado que los GRNG basados en el TCL no muestran una precisión elevada en la cola de la distribución [28]. Sin embargo, en el caso de la inferencia de las BNN no es necesario tener un GNRG preciso. Hirayama *et al.* demostraron que utilizando un GRNG de poca precisión basado en una *Lookup Table* (LUT) se obtienen buenos resultados [29].

En otro trabajo Awano *et al.* diseñaron un acelerador que aproximan la salida final del modelo utilizando generadores hardware de muestras Bernoulli, bajo la condición de que el TCL podría aplicarse al modelo en general [25]. La Figura 2.4 muestra un diagrama de su aproximación.

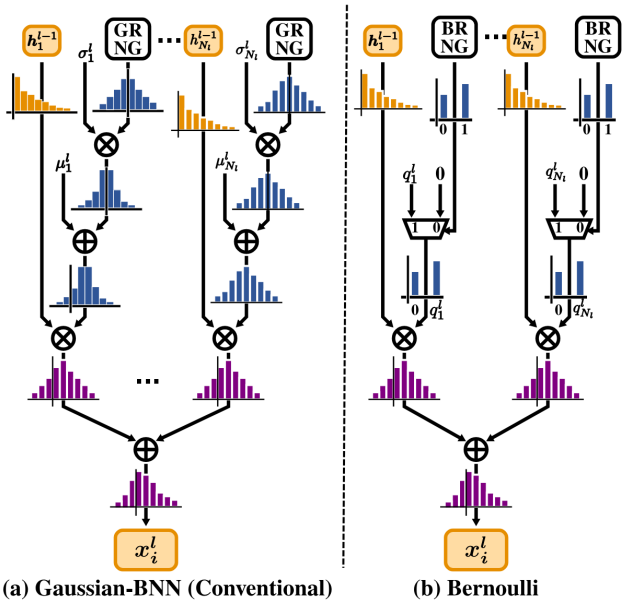


Figura 2.4: Comparación de muestrear distribuciones gaussianas o distribuciones de Bernoulli en una BNN. Debido al TCL la distribución final es independiente de las distribuciones muestreadas [25].

Una ventaja clave de los generadores de muestras Bernoulli radica en su implementación en hardware. A diferencia de los métodos de software que dependen de operaciones de salto, que pueden degradar el rendimiento de las CPU modernas, los generadores hardware pueden implementarse utilizando componentes simples como comparadores y multiplexores, lo que los hace de bajo coste y eficientes.

2.2.3. Soporte de bibliotecas

Este trabajo tiene como objetivo ejecutar inferencia de estas redes en entornos RISC-V de bajas prestaciones, donde actualmente no existe un conjunto de bibliotecas que lo permita. TensorFlow Probability permite entrenar y ejecutar inferencia de BNN utilizando precisión en punto flotante [30]. Por otro lado, en el caso de PyTorch, existe BayesianTorch desarrollado por IntelLabs, que ofrece la misma funcionalidad [31].

TensorFlowLite es la versión de TensorFlow diseñada para sistemas empotrados, ofrece funcionalidades como la cuantización de modelos y la inferencia utilizando precisión de enteros para las NN convencionales, sin embargo, aún no tiene soporte para BNN [32]. En una actualización reciente, BayesianTorch ha añadido soporte para la cuantización de BNN [33]. Este trabajo se centra en el ecosistema de Tensorflow debido a que se han utilizado modelos desarrollados en dicho entorno para su validación.

Capítulo 3

Metodología

3.1. Herramientas utilizadas

La Tabla 3.1 muestra un listado de todas las herramientas utilizadas para el desarrollo de este trabajo junto con sus versiones y descripciones de uso.

Tabla 3.1: Herramientas utilizadas junto con sus versiones y descripciones de uso.

Nombre	**Versión**	**Descripción**
Git [34]	2.45.1	Herramienta de control de versiones
GitHub [35]	-	Repositorio de código en línea
riscv vhdl [36]	Commit	Repositorio de código del procesador RISC-V base
BNN for hyperspectral datasets analysis [37]	Commit	Repositorio de código para entrenar y utilizar BNN para clasificar píxeles hiperespectrales
VHDL	2008	Lenguaje de descripción hardware utilizado para definir el procesador RISC-V y su extensión
GDHL LLVM [38]	5.0.0	Simulador de VHDL con backend de LLVM
GTKWave [39]	3.3.118	Visor de ondas utilizado para depurar los circuitos diseñados en VHDL
C	C11	Lenguaje de programación utilizado para desarrollar los programas que se han ejecutado en los procesadores simulados
GNU RISC-V Compiler Toolchain [40]	12.2.0	Conjunto de herramientas para compilar programas C para plataformas RISC-V
Python	3.11.9	Lenguaje de *scripting* utilizado para desarrollar varios componentes de este trabajo además de para automatizar procesos de compilación y pruebas
TensorFlow Probability [30]	0.20.1	Biblioteca de Python para trabajar con BNN
pyelftools [41]	0.31	Biblioteca de Python para trabajar con ficheros ELF
Matplotlib [42]	3.7.1	Biblioteca de Python para crear gráficos
SciPy [43]	1.10.1	Biblioteca de Python con utilidades estadísticas
Make	4.4.1	Herramienta de automatización de compilación y pruebas

Bash	5.2.26	Herramienta de automatización de compilación y pruebas
Xilinx ZCU104 [44]	-	Plataforma de desarrollo FPGA utilizada
Vivado Design Suite [45]	2023	Conjunto de herramientas para implementar diseños hardware descritos en VHDL en FPGA y analizar su coste
Overleaf [46]	-	Editor de LaTeX en línea utilizado para la redacción de este documento
draw.io [47]	-	Editor de diagramas en línea
Inkscape [48]	1.3.2	Editor de gráficos vectoriales

3.2. Modelos utilizados

Durante el proceso de verificación se han utilizado tres arquitecturas de modelos diferentes. Estos modelos se han elegido porque han sido utilizados por otros trabajos recientes o porque son arquitecturas estándar muy conocidas. La Tabla 3.2 muestra la arquitectura de los modelos para clasificación de píxeles hiperespectrales [18]. Los conjuntos de datos utilizados para este modelo son diferentes imágenes hiperespectrales obtenidas mediante satélite denominadas BO, IP, KSC, PU y SV.

Tabla 3.2: Arquitectura de los modelos para clasificación de píxeles hiperespectrales.

Tipo de capa	**Entrada**	**Salida**
Dense ReLU	Número de bandas espectrales	32
Dense ReLU	32	16
Dense SoftMax	16	Número de clases de píxeles

La Tabla 3.3 muestra una arquitectura LeNet-5 bayesiana. LeNet-5 es una arquitectura de red neuronal convolucional (**C***onvolutional* **N***eural* **N***etwork*) simple [49]. Se ha entrenado un modelo con la misma arquitectura para reconocer el conjunto de datos MNIST [50] y CIFAR-10 [51] pero utilizando capas bayesianas.

Tabla 3.3: Arquitectura LeNet-5 bayesiana.

Tipo de capa	**Entrada**	**Salida**	**Tamaño del filtro**
Conv2D Valid ReLU	28×28	24×24×6	5×5
MaxPool2D	24×24×6	12×12×6	2×2
Conv2D Valid ReLU	12×12×6	8×8×16	5×5
MaxPool2D	8×8×16	4×4×16	2×2
Conv2D Valid ReLU	4×4×16	1×1×120	4×4
Dense ReLU	120	84	
Dense SoftMax	84	10	

La Tabla 3.4 muestra la arquitectura B2N2, propuesta por Awano *et al.* junto a su acelerador para BNN [25]. También se ha entrenado para reconocer los conjuntos de datos MNIST y CIFAR-10.

Tabla 3.4: Arquitectura B2N2.

Tipo de capa	**Entrada**	**Salida**	**Tamaño del filtro**
Conv2D Same ReLU	32×32	32×32×32	3×3
Conv2D Same ReLU	32×32×32	32×32×32	3×3
MaxPool2D	32×32×32	16×16×32	2×2
Conv2D Same ReLU	16×16×32	16×16×64	3×3
Conv2D Same ReLU	16×16×64	16×16×64	3×3
MaxPool2D	16×16×64	8×8×64	2×2
Conv2D Same ReLU	8×8×64	8×8×128	3×3
Conv2D Same ReLU	8×8×128	8×8×128	3×3
MaxPool2D	8×8×128	4×4×128	2×2
Dense SoftMax	2028	10	

3.3. Configuración experimental

La Figura 3.1 muestra el proceso y componentes para ejecutar inferencia de BNN de manera eficiente en un procesador RISC-V con soporte único para precisión entera que se han utilizado para obtener los resultados de este trabajo.

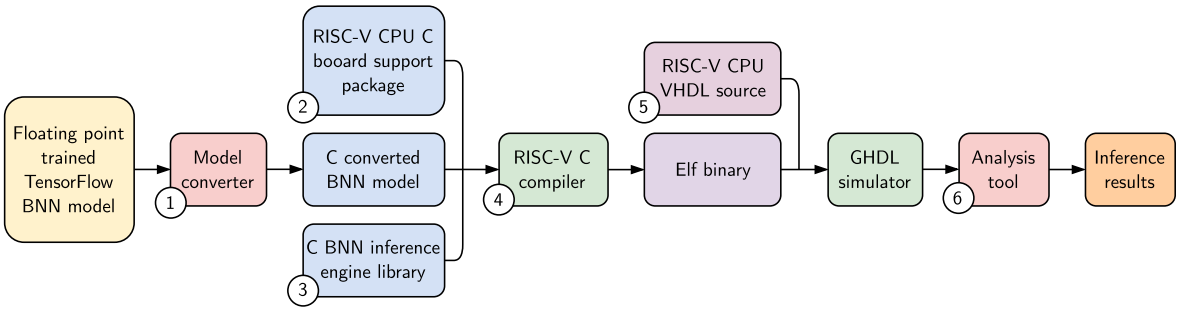


Figura 3.1: Diagrama de componentes necesarios para ejecutar la inferencia de BNN en un procesador RISC-V simulado. Los componentes marcados con un número son contribuciones de este trabajo.

3.3.1. Conversor de modelos

Se ha desarrollado una herramienta en Python, marcada en la Figura 3.1 como componente 1, que convierte modelos BNN TensorFlow ya entrenados en precisión de punto flotante a ficheros de código C con precisión en coma fija que puedan ejecutarse junto al motor de inferencia explicado en el Capítulo 4. La Figura 3.2 muestra un diagrama de las diferentes operaciones que realiza esta herramienta.

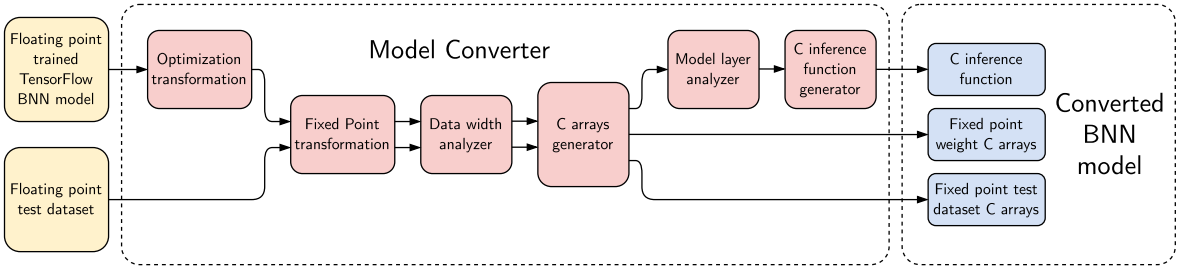


Figura 3.2: Diagrama de componentes internos del conversor de modelos.

Primero aplica transformaciones a los pesos requeridas por las optimizaciones detalladas en el Capítulo 5 utilizando coma flotante para mantener la máxima precisión posible, tras ello codifica los valores en coma fija utilizando una escala constante previamente definida. A continuación analiza el número de bits necesarios para almacenar los datos y selecciona los datos nativos de C con los bits necesarios, seguidamente almacena todos los pesos en vectores de C. Finalmente analiza la función de inferencia del modelo y crea una versión en C utilizando las funciones proporcionadas por el motor de inferencia.

Precisión en coma fija

Implementar aritmética de coma flotante en hardware es caro, por lo que es común que procesadores de bajas prestaciones no dispongan de dicho hardware. GCC permite emular las operaciones de coma flotante mediante software, comúnmente conocido como *soft-float*. Esta emulación tiene un gran impacto en el rendimiento por lo que no se ha utilizado.

Para mejorar el rendimiento, se ha adoptado el formato de coma fija para la representación de números decimales. Este formato permite realizar operaciones con números decimales utilizando hardware de precisión entera, lo que tiene un impacto mínimo en el rendimiento. Esta codificación consiste en multiplicar los números con una escala potencia de 2. A mayor sea la escala mayor precisión se obtiene. La principal limitación de esta codificación es el tamaño de palabra de la arquitectura ya que pueden ocurrir desbordamientos al operar con números multiplicados por una escala grande. Por lo que en general se obtiene menor precisión que con la codificación en punto flotante. La Figura 3.3 muestra un ejemplo de este tipo de codificación.

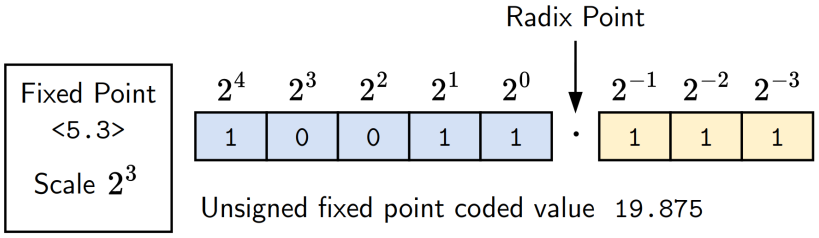


Figura 3.3: Ejemplo de codificación en coma fija sin signo <5.3> del valor 19.875.

3.3.2. Actualización del Board Support Package

El **B***oard* **S***upport* **P***ackage* (BSP) de una plataforma es una capa de software intermedia entre la aplicación y el hardware que contiene código específico para crear un entorno de ejecución estable en dicha plataforma. Además puede proveer de los ficheros de configuración y un sistema de compilación. En la Figura 3.1 aparece como componente 2. En la Figura 3.4 se muestra un diagrama con más detalle sobre este componente.

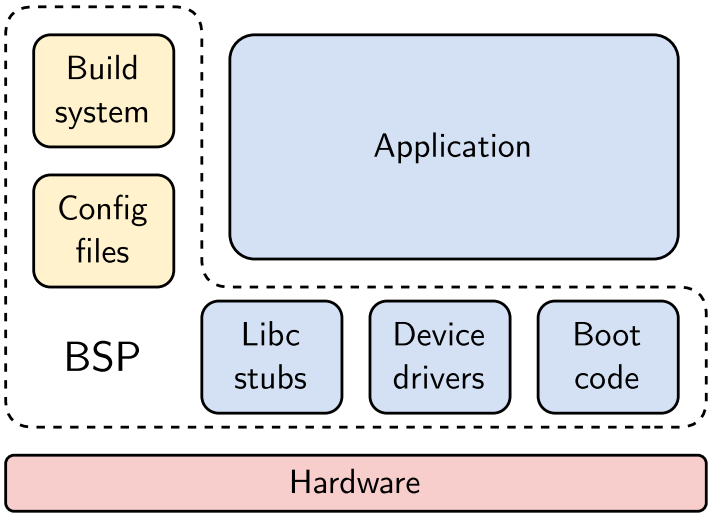


Figura 3.4: Diagrama de componentes que forman un BSP y cómo se sitúa entre la aplicación y el hardware.

Con el objetivo de mejorar la experiencia de desarrollo del motor de inferencia se actualizó el BSP del procesador para dar soporte a la biblioteca estándar C (libc) y algunas funcionalidades de C++. De esta forma se pueden utilizar funciones útiles de libc cómo `printf`, o `memset` entre otras. Para ello se actualizaron las herramientas y ficheros de compilación, se añadieron funciones *stub* para las llamadas al sistema no implementadas y se implementaron versiones modificadas de `_write` y `_exit`.

3.3.3. Verificación

Para validar el correcto funcionamiento del motor de inferencia y estudiar el impacto en el rendimiento de las optimizaciones posteriores se ha desarrollado una herramienta que compara las predicciones del conjunto de datos de prueba obtenidas con el motor de inferencia y TensorFlow. Las predicciones obtenidas con TensorFlow se han tomado como punto de referencia correcto. Este componente aparece en la Figura 3.1 como componente 6.

Para comparar los conjuntos de predicciones se han utilizado las métricas de incertidumbre y la precisión. La precisión se puede calcular como el ratio de predicciones acertadas sobre el número total de predicciones. Analizar las métricas de incertidumbre y

las propiedades estadísticas de los modelos es complejo. Para hacerlo se han utilizado 4 tipos de gráficas que las representan, estas se explicarán más adelante en la Sección 4.2 mediante ejemplos.

Capítulo 4

Motor de inferencia para Redes Neuronales Bayesianas

4.1. Biblioteca desarrollada en C

Cómo las bibliotecas estándar no ofrecen soporte para la inferencia de BNN en plataformas de bajas prestaciones, en consecuencia, se ha desarrollado una biblioteca de funciones que ejecutan la inferencia de capas BNN estándar y convolucionales con funciones de activación ReLU o exponenciales normalizadas (SoftMax) utilizando precisión de coma fija. Esta biblioteca se ha desarrollado en C y se muestra en la Figura 3.1 como el componente número 3. A continuación, se describe brevemente las principales decisiones de diseño.

4.1.1. Aproximación de la función logaritmo

Para calcular las métricas de incertidumbre se utiliza la función logaritmo por lo que se ha implementado una versión del algoritmo desarrollado por Turner para calcular el logaritmo de un número en coma fija [52].

4.1.2. Aproximación de la función SoftMax

La función de activación SoftMax se utiliza en la última capa de una NN para transformar su vector de salida de forma que la suma de todos sus componentes sea 1, lo que facilita su interpretación. Mientras que calcular la función de activación ReLU es trivial, la función SoftMax no, por lo que una de las contribuciones de este trabajo ha sido crear una aproximación de esta función. La función SoftMax toma un vector de componentes $x \in X$ de tamaño N como entrada y devuelve otro del mismo tamaño cuyos componentes $y \in Y$ se calculan según la Ecuación 4.1.

$$y_i = \frac{e^{x_i}}{\sum_{j=0}^{N} e^{x_j}} \tag{4.1}$$

Para calcular esta función utilizando coma fija es necesario calcular la función exponencial en dicho formato. Para evitar desbordamientos de la función exponencial se va a utilizar la función SoftMax equivalente mostrada en la Ecuación 4.2.

$$y_i = \frac{e^{x_i - \text{máx}(X)}}{\sum_{j=0}^{N} e^{x_j - \text{máx}(X)}} \tag{4.2}$$

Como se cumple que $x_i - \text{máx}(X) \in (-\infty, 0]$ entonces se cumple que $e^{x_i} \in (0, 1]$. Para calcular la función exponencial en dicho rango se va a utilizar la Ecuación 4.3, dividiendo la entrada x_i en su parte entera a_i y su parte decimal b_i.

$$e^{x_i} = e^{a_i + b_i} = e^{a_i} e^{b_i} \tag{4.3}$$

La parte entera e^{a_i} se calcula mediante una LUT de 20 entradas para el rango $[e^{-19}, e^{0}]$. A partir de e^{-19} los valores son demasiado pequeños como para representarlos con la precisión disponible por lo que siempre toman el valor 0. La parte decimal e^{b_i} se aproxima mediante los 8 primeros términos de la serie de Taylor mostrada en la Ecuación 4.4. Para optimizar su cálculo evitando las instrucciones de división, los valores de $\frac{1}{n!}$ para $n \in [2, 7]$ se han almacenado en una LUT.

$$e^{b_i} \approx \sum_{n=0}^{7} \frac{{b_i}^n}{n!} \tag{4.4}$$

La Figura 4.1 muestra un estudio de la precisión de la aproximación. Se comparan sus resultados con los de la implementación de la función exponencial de la biblioteca estándar en punto flotante. Como medida cuantitativa del error se ha utilizado el error cuadrático medio (**M***ean* **S***quared* **E***rror*). Se aprecia cómo se obtiene un MSE muy bajo en el rango deseado, lo que implica que es una buena aproximación.

4.1.3. Muestreo de distribuciones Gaussianas

Como se ha explicado previamente, las BNN necesitan muestrear distribuciones Gaussianas. Como método de muestreo base se ha utilizado un algoritmo basado en la suma de distribuciones uniformes, dicha suma se aproxima a una distribución gaussiana debido al TCL, como se muestra en la Ecuación 4.5.

$$\sum_{n=0}^{N} \mathcal{U}_n(0, 1) \sim \mathcal{N}\left(\frac{N}{2}, \sqrt{\frac{N}{12}}\right) \tag{4.5}$$

El algoritmo implementado genera muestras de $\mathcal{N}(0, 1)$ para luego transformarlas en muestras de una distribución gaussiana arbitraria $\mathcal{N}(\mu, \sigma)$ de media μ y desviación típica

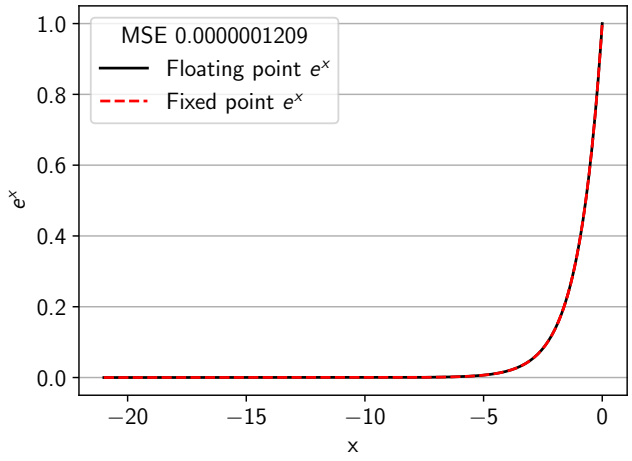


Figura 4.1: Comparación de la función e^x de la biblioteca estándar en punto flotante (línea negra) con la aproximación en coma fija implementada (línea roja discontinua) en el rango $[-20, 0]$.

σ utilizando la Ecuación 4.6.

$$\mathcal{N}(\mu, \sigma) = \sigma\mathcal{N}(0, 1) + \mu \tag{4.6}$$

La aproximación empleada utiliza la suma de 12 distribuciones uniformes de forma que la desviación típica de la distribución resultante sea 1, lo que permite centrar la distribución como muestra la Ecuación 4.7.

$$\sum_{n=0}^{12} \mathcal{U}_n(0, 1) - 6 \sim \mathcal{N}(0, 1) \tag{4.7}$$

La Figura 4.2 muestra un análisis estadístico del método de muestreo implementado. En el gráfico Q-Q se aprecia cómo los cuantiles del conjunto de muestras son muy parecidos a los cuantiles teóricos, mostrando desviaciones en las colas cómo es de esperar de esta aproximación. En el histograma se aprecia cómo las muestras se ajustan bien a la distribución teórica.

4.1.4. Muestreo de distribuciones Uniformes

Generar muestras de distribuciones uniformes es una parte del algoritmo para generar muestras de una distribución gaussiana explicado previamente. Para hacerlo se utiliza una versión del algoritmo Xorshift de 32 bits [53]. Es un algoritmo sencillo y de bajo coste para generar números pseudoaleatorios uniformes solamente mediante instrucciones `xor` y `shift`.

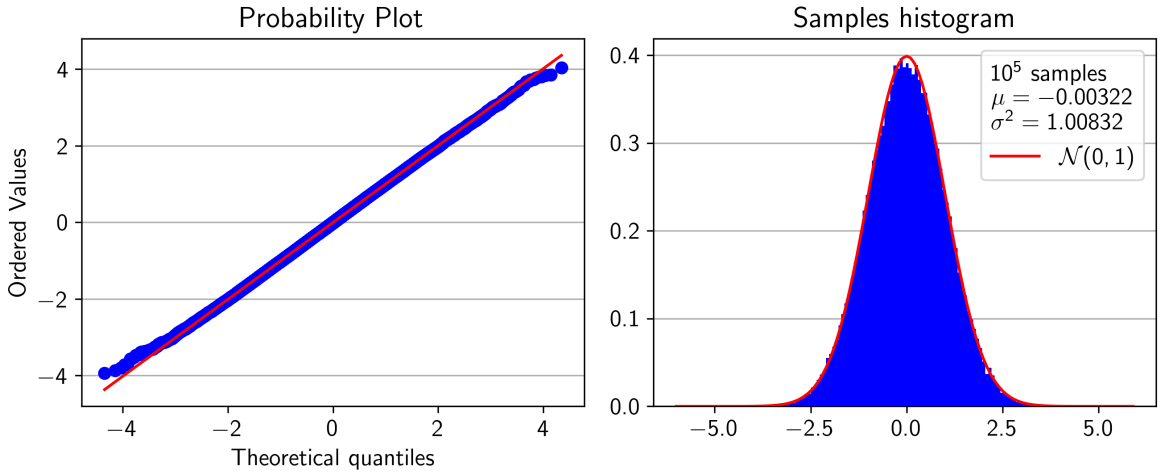


Figura 4.2: Análisis estadístico de 10^5 muestras del GRNG implementado (azul) con respecto a $\mathcal{N}(0, 1)$ (rojo). A la izquierda se muestra un gráfico Q-Q. A la derecha se muestra un histograma.

4.2. Análisis de resultados

La Tabla 4.1 muestra la precisión obtenida con los diferentes modelos utilizando ambos motores de inferencia. Al tratarse de modelos probabilísticos pequeñas fluctuaciones son esperables, especialmente en las predicciones con mucha incertidumbre.

Tabla 4.1: Comparación de precisión obtenida con el motor de inferencia implementado con respecto a resultados obtenidos con TensorFlow

	Motor de inferencia	
Modelo	*TensorFlow*	*C*
Hiperespectral BO	0.9039	0.9101
Hiperespectral IP	0.8139	0.8148
Hiperespectral KSC	0.9256	0.9217
Hiperespectral PU	0.9017	0.9021
Hiperespectral SV	0.9257	0.9275
Lenet-5 MNIST	0.9836	0.9830
Lenet-5 CIFAR10	0.6351	0.6229
B2N2 MNIST	0.9872	0.9854
B2N2 CIFAR10	0.7295	0.7134

La Figura 4.3 muestra un ejemplo de las gráficas utilizadas para validar las métricas de incertidumbre y propiedades estadísticas. Estas gráficas se han creado utilizando las predicciones para el conjunto de datos de píxeles hiperespectrales KSC.

La Subfigura 4.3a muestra un histograma de la incertidumbre agrupada por aciertos y fallos. La Subfigura 4.3b muestra la incertidumbre media, separada en epistémica y aleatoria, de las predicciones de cada clase. La Subfigura 4.3c muestra la precisión con respecto a la incertidumbre junto con un histograma de los datos agrupados también con respecto a la

incertidumbre. Y finalmente, la Subfigura 4.3d muestra la recta de calibración del modelo.

Como se muestra en el ejemplo se obtienen resultados muy similares en todas las gráficas. Las únicas diferencias notables se aprecian en las rectas de precisión de la Figura 4.3c, pero estas diferencias ocurren en predicciones con incertidumbre alta de las que además hay muy pocas. El modelo al reportar una incertidumbre elevada ya está avisando de la baja fiabilidad de la predicción con lo que las diferencias en este tipo de predicciones son esperables y aceptables. Para el resto de modelos se obtienen resultados con las mismas similitudes entre motores de inferencia, se pueden consultar todos en el Anexo A.

En conjunto, los resultados demuestran que el motor de inferencia desarrollado no perjudica ni las métricas de incertidumbre ni la precisión de los modelos incluso tras transformar los datos a coma fija perdiendo precisión en la representación.

4.3. Análisis de la carga de trabajo

La principal operación en la inferencia de NN es **M***ultiply*–**AC***cumulate* (MAC), mostrada en la Ecuación 4.8. Una capa es un conjunto de neuronas, las entradas a una neurona x_i se multiplican por sus pesos w_i y se acumulan junto a al *bias* b. Una capa no convolucional se puede representar como una única multiplicación de matriz por vector.

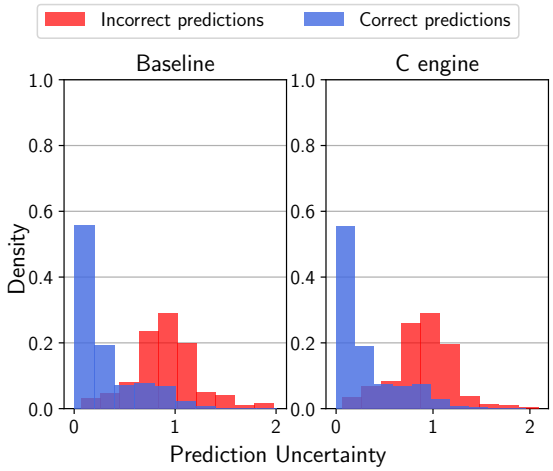
$$b + \sum_{i=0}^{N} w_i x_i \tag{4.8}$$

En el caso de las BNN estas operaciones MAC además requieren muestrear una distribución Gaussiana, como se muestra en la Ecuación 4.9.

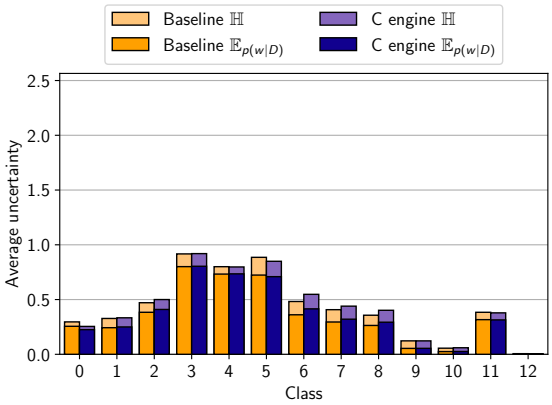
$$b + \sum_{i=0}^{N} Sample(\mu_i, \sigma_i)\ x_i \tag{4.9}$$

El procesador RISC-V dispone de un contador de ciclos llamado `mcycle`, lo que permite medir prestaciones con precisión a nivel de ciclo. Se ha utilizado este contador para crear un perfil de la carga de trabajo del motor de inferencia, el cual se muestra en la Figura 4.4.

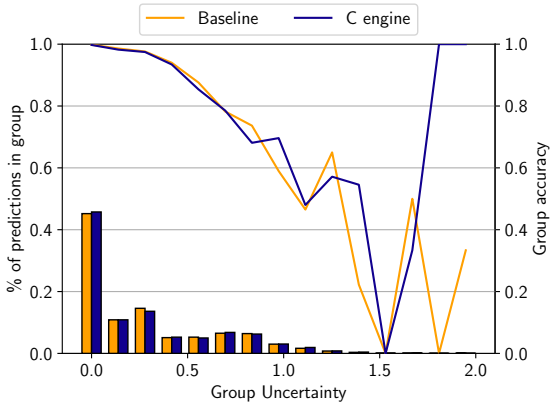
La operación de muestreo es la más costosa con diferencia, ocupando más del 80 % de los ciclos de ejecución en los 3 modelos diferentes. Por ello, uno de los objetivos principales de este trabajo y de otros relacionados ha sido optimizar esta operación.



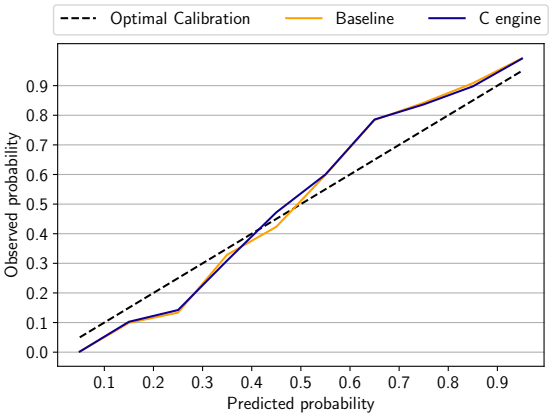
(a) Histogramas de incertidumbre divididos en predicciones incorrectas (rojo) y predicciones correctas (azul).



(b) Incertidumbre predictiva ($\mathbb{H}$) y aleatoria ($\mathbb{E}p$) agrupada por clases.



(c) Precisión (líneas) y porcentaje de predicciones (barras) agrupadas por incertidumbre.



(d) Recta de calibración del modelo, muestra probabilidades observadas con respecto a las probabilidades predichas por el modelo.

Figura 4.3: Ejemplos de gráficas para analizar la incertidumbre y propiedades estadísticas de un modelo BNN. En 4.3b, 4.3c y 4.3d los colores amarillos representan los resultados obtenidos con TensorFlow y los colores azules los obtenidos con el motor de inferencia desarrollado. Predicciones del conjunto de prueba de píxeles hiperespectrales KSC.

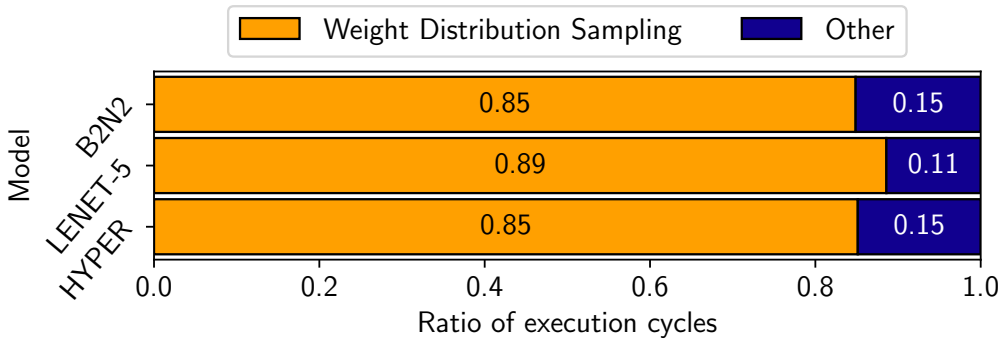


Figura 4.4: Ratio de ciclos de ejecución dedicados al muestreo de distribuciones gaussianas (amarillo) y al del resto de operaciones (azul) en las diferentes arquitecturas de modelos utilizadas.

Capítulo 5

Optimizaciones software para acelerar el muestreo de distribuciones

A continuación se presentan 2 propuestas de aceleración software con distribuciones Bernoulli y Uniformes como candidatas a sustituir a distribuciones Gaussianas, ya que muestrear estas últimas es más costoso.

5.1. Distribuciones Bernoulli

Awano *et al.* propusieron muestrear distribuciones Bernoulli en vez de distribuciones Gaussianas para aumentar la eficiencia del algoritmo en un acelerador para inferencia [25]. Los parámetros de estas nuevas distribuciones se obtienen mediante una transformación de los parámetros de las distribuciones originales. En este trabajo se ha implementado esta optimización mediante software y se ha analizado su impacto en la precisión, métricas de incertidumbre y rendimiento. A continuación se explican los fundamentos teóricos de la optimización.

El TCL expone que en condiciones generales la distribución de una suma de variables aleatorias tiende a ser una distribución gaussiana. Las operaciones MAC son una suma de variables aleatorias por lo que según el TCL el resultado seguirá una distribución gaussiana. Una distribución gaussiana se define con 2 parámetros, la media y la desviación típica. Asumiendo que el tipo de distribución resultante es independiente del tipo de distribuciones de los sumandos, solamente se tiene que preservar la esperanza y la varianza de la operación MAC para no alterar los resultados de la BNN. Las Ecuaciones 5.1 y 5.2 muestran la esperanza y la varianza de una operación MAC.

$$E\left[b + \sum_{i=0}^{N} w_i x_i\right] = b + \sum_{i=0}^{N} (E[w_i]E[x_i]) \tag{5.1}$$

$$V\left[b + \sum_{i=0}^{N} w_i x_i\right] = \sum_{i=0}^{N} (V[w_i]V[x_i] + V[w_i]E[x_i]^2 + V[x_i]E[w_i]^2) \tag{5.2}$$

La optimización consiste en utilizar un nuevo peso w' que cumpla $E[w'] = E[w]$ y $V[w'] = V[w]$, de forma que no altere los resultados de la BNN. Se define $w' = qb$, siendo b una muestra de una distribución Bernoulli $\mathcal{B}(p)$, esta distribución toma el valor 1 con probabilidad p y el valor 0 con probabilidad $1 - p$. Las constantes p y q se calculan a partir de la media μ y desviación típica σ de las distribuciones originales mediante el sistema de ecuaciones mostrado en la Ecuación 5.3.

$$\begin{cases} \mu = pq \\ \\ \sigma^2 = p(1-p)q^2 \end{cases} \rightarrow \begin{cases} p = \dfrac{\mu^2}{\mu^2 + \sigma^2} \\ \\ q = \dfrac{\mu^2 + \sigma^2}{\mu} \end{cases} \tag{5.3}$$

Esta optimización no aumenta el tamaño de los modelos y no necesita muestrear distribuciones gaussianas sino Bernoulli, cuyo algoritmo de muestreo es mucho más sencillo. Dicho algoritmo de muestreo incluye una instrucción de salto, este tipo de instrucciones puede tener un coste muy elevado en algunas arquitecturas hardware. Por lo que este trabajo ha desarrollado otra optimización que utiliza una distribución que no requiere este tipo de instrucciones, explicada a continuación.

5.2. Distribuciones Uniformes

Partiendo de Awano *et al.* este trabajo propone un nuevo método que utiliza distribuciones Uniformes en vez de distribuciones Bernoulli. En este caso el nuevo peso se define como $w' = bu + a$, siendo u una muestra de una distribución Uniforme $\mathcal{U}(0, 1)$. a y b siendo constantes que se pueden calcular a partir de la media μ y desviación típica σ de los pesos originales utilizando el sistema de ecuaciones mostrado en la Ecuación 5.4.

$$\begin{cases} \mu = \dfrac{b}{2} + a \\ \\ \sigma^2 = \dfrac{b^2}{12} \end{cases} \rightarrow \begin{cases} a = \mu - \dfrac{b}{2} \\ \\ b = \sigma\sqrt{12} \end{cases} \tag{5.4}$$

Esta optimización tampoco aumenta el tamaño de los modelos, muestrea distribuciones Uniformes cuyo algoritmo de muestreo es muy sencillo y además no requiere instrucciones de salto.

5.3. Análisis de resultados

La Tabla 5.1 muestra los resultados de precisión y aceleración (*speedup*) obtenidos con las optimizaciones Bernoulli y Uniforme para todas las arquitecturas de modelos y conjuntos de datos.

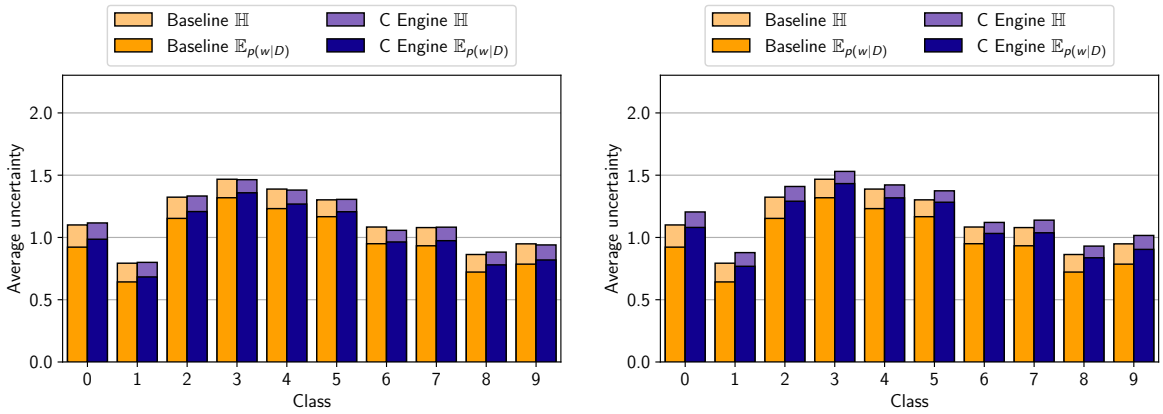
Tabla 5.1: Resultados de precisión y *speedup* obtenidos con las optimizaciones Bernoulli y Uniforme para todas las arquitecturas de modelos y conjuntos de datos.

Modelo	**Conjunto de datos**	**Precisión**			**Speedup**	
		TensorFlow	*Bernoulli*	*Uniforme*	*Bernoulli*	*Uniforme*
HYPER	BO	0.9039	0.9039	0.9082	5.18	4.95
	IP	0.8139	0.8154	0.8130		
	KSC	0.9256	0.9274	0.9217		
	PU	0.9017	0.9004	0.9016		
	SV	0.9257	0.9238	0.9279		
LENET-5	MNIST	0.9836	0.9883	0.9885	4.28	3.87
	CIFAR-10	0.6351	0.6336	0.6220		
B2N2	MNIST	0.9872	0.9917	0.9910	5.67	5.64
	CIFAR-10	0.7295	0.7273	0.7197		

Ninguna de las optimizaciones tiene un efecto negativo en la precisión de los modelos utilizados para verificación. Si el muestreo de la distribución Bernoulli devuelve 0 se pueden ignorar el resto de instrucciones de la operación MAC de dicho operando, ya que acumularía 0. Esto lo hace el compilador de forma automática en la fase de optimización de eliminación de código muerto. En el caso del procesador RISC-V utilizado la penalización de las instrucciones de salto es solamente de 2 ciclos, por lo que debido a esto junto con la eliminación de código la optimización Bernoulli obtiene un *speedup* mayor que la Uniforme.

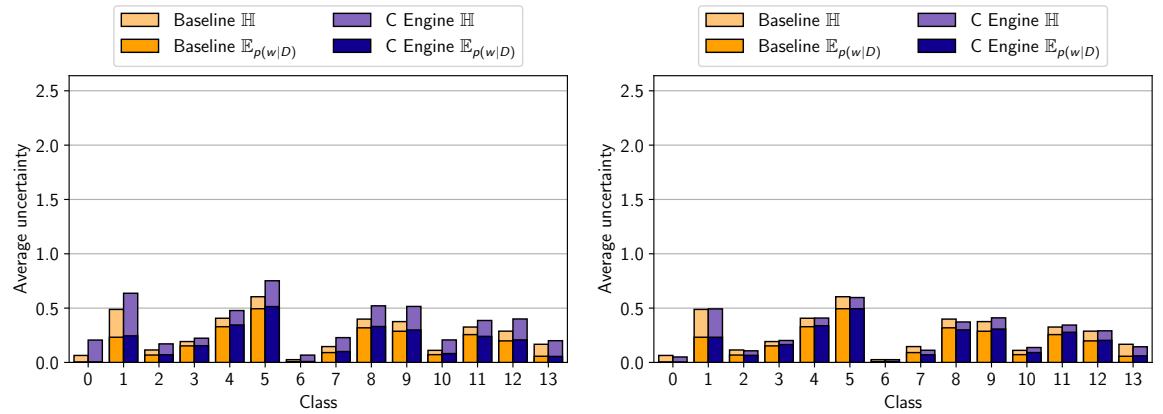
Con respecto a la preservación de las métricas de incertidumbre, ambas optimizaciones producen pequeñas alteraciones. La magnitud de estas alteraciones es dependiente de la optimización y del modelo, en algunos casos la optimización Bernoulli produce alteraciones menores y en otros la Uniforme. La Figura 5.1 muestra un caso en el que la optimización Bernoulli produce menos alteraciones que la Uniforme y la Figura 5.2 un caso en el que ocurre lo contrario. Se ha considerado que estas variaciones son pequeñas y no llegan a tener un efecto negativo en la utilidad de las métricas.

En el Anexo B se pueden encontrar los resultados obtenidos con todos los modelos y conjuntos de datos utilizando la optimización Bernoulli y en el Anexo C utilizando la optimización Uniforme.



(a) Optimización Bernoulli (b) Optimización Uniforme

Figura 5.1: Incertidumbre predictiva ($\mathbb{H}$) y aleatoria ($\mathbb{E}p$) agrupada por clases de las predicciones obtenidas con *TensorFlow* (amarillo) y el motor de inferencia optimizado (azul) del modelo LeNet-5 y el conjunto de datos CIFAR-10.



(a) Optimización Bernoulli (b) Optimización Uniforme

Figura 5.2: Incertidumbre predictiva ($\mathbb{H}$) y aleatoria ($\mathbb{E}p$) agrupada por clases de las predicciones obtenidas con *TensorFlow* (amarillo) y el motor de inferencia optimizado (azul) del conjunto de datos de píxeles hiperespectrales BO.

Capítulo 6

Extensión RISC-V para inferencia de Redes Neuronales Bayesianas

6.1. Diseño de una nueva unidad funcional

Como se ha explicado en secciones previas, la sección del algoritmo más importante a optimizar es el muestreo de distribuciones Gaussianas. Para ello se va incluir un GRNG como una nueva UF en la CPU RISC-V base. Sarwar Malik *et al.* propusieron un diseño basado en el TCL que añadía un componente corrector para reducir el error de muestreo de las colas de la distribución [28]. Como también se ha mostrado previamente, no es necesaria una gran precisión a la hora de muestrear distribuciones, por lo que se va a diseñar un GNRG basado en la suma de 12 distribuciones uniformes sin bloque corrector.

Se ha optado por utilizar un GRNG y no un generador de otro tipo como el que requieren alguna de las optimizaciones software para desviarse lo menos posible del algoritmo original, además de crear una UF que pueda ser útil para otro tipo de aplicaciones que requieran muestrear distribuciones Gaussianas, como por ejemplo la eliminación de ruido en procesamiento de señal.

6.1.1. Generador de números pseudoaleatorios Uniformes

Para generar números pseudoaletorios uniformes se ha utilizado un componente hardware llamado **L***inear* **F***eedback* **S***hift* **R***egister* (LFSR). Se basan en un circuito de retroalimentación y un registro de estado. El circuito de retroalimentación es un conjunto de puertas XOR que implementan un polinomio generador. Dado un registro de estado de n bits, si el polinomio puede producir $2^n - 1$ valores antes de empezar a repetir la secuencia entonces la secuencia es máxima. Alfke recopiló una lista de polinomios generadores de secuencias máximas para varios tamaños de registros de estado [54]. La desventaja de este tipo de LFSR es que no pueden generar más de un bit con baja correlación entre muestras. La Figura 6.1 muestra diagramas de muestras con alta correlación, el diagrama

de autocovarianza debería mostrar un único pico en el 0 y el gráfico de dispersión un patrón de ruido blanco.

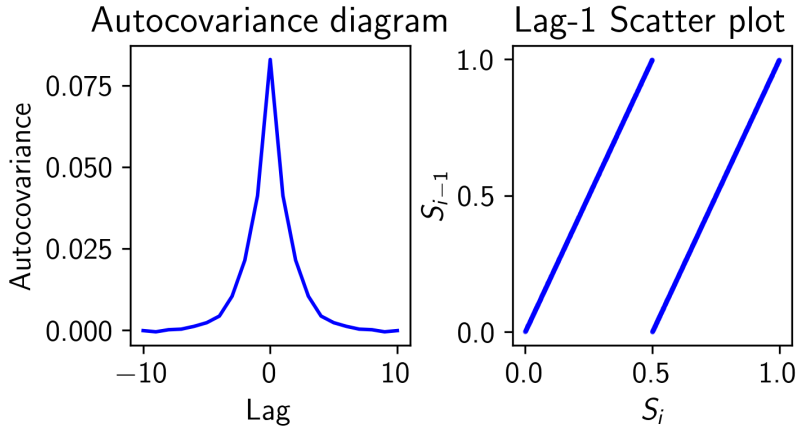


Figura 6.1: Diagramas de correlación de 10^4 muestras de 12 bits de un LFSR. A la izquierda un diagrama de autocovarianza para diferentes distancias entre muestras. A la derecha un diagrama de dispersión de una muestra i con respecto a la $i-1$.

Para paliar este problema se ha utilizado un *Lookahead* LFSR [55]. Estos LFSR tienen un circuito de retroalimentación más complejo que generan n bits con baja correlación. Este circuito aplica el polinomio generador base n veces, Colavito *et al.* detallaron como obtener estos circuitos mediante exponenciación de matrices y que restricciones deben seguir [56]. En este trabajo se ha desarrollado un *script* en Python que genera código VHDL para implementar *Lookahead* LFSR utilizando su método. La Figura 6.2 muestra los diagramas de correlación que se obtienen utilizando este tipo de LFSR.

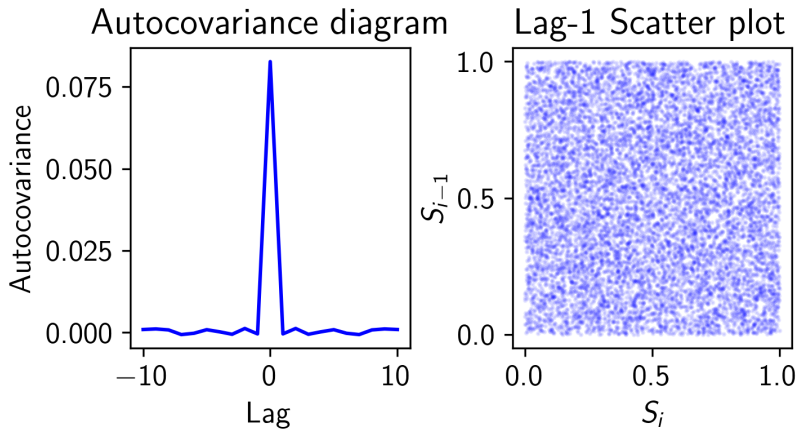


Figura 6.2: Diagramas de correlación de 10^4 muestras de 12 bits de un 12-*Lookahead* LFSR. A la izquierda un diagrama de autocovarianza para diferentes distancias entre muestras. A la derecha un diagrama de dispersión de una muestra i con respecto a la $i-1$.

6.1.2. Generador de números pseudoaleatorios Gaussianos

Un GRNG basado en el TCL tiene 2 componentes principales, un generador de muestras de distribuciones uniformes y un árbol de sumadores que las acumula. Para sumar 12 muestras se necesita un árbol de 4 niveles. Como se busca obtener una muestra final de 16 bits las muestras iniciales deben ser de 12 bits, ya que por cada nivel del árbol las muestras aumentan su tamaño en 1 bit para evitar desbordamientos. Para evitar afectar negativamente a la frecuencia del reloj de la CPU original el árbol se ha segmentado por niveles, lo que sigue permitiendo obtener una muestra por ciclo sin penalizar a la frecuencia del reloj a cambio de aumentar el coste con registros de estado intermedios. Esta decisión penaliza en el cambio de semilla, que obliga a esperar 4 ciclos para obtener muestras con la semilla actualizada. La actualización de la semilla del generador no es una operación común por lo que no se ha considerado un problema.

Para generar 12 muestras de 12 bits se utiliza un 144-*Lookahead* LFSR con un registro de estado de 151 bits. Las muestras representan valores entre 0 y 1, codificados en coma fija con una escala 2^{12}. Para centrar la muestra final se utiliza un restador de 16 bits con un valor constante $6 \cdot 2^{12}$. La Figura 6.3 muestra un diagrama del diseño y la Figura 6.4 la máquina de estados de su unidad de control.

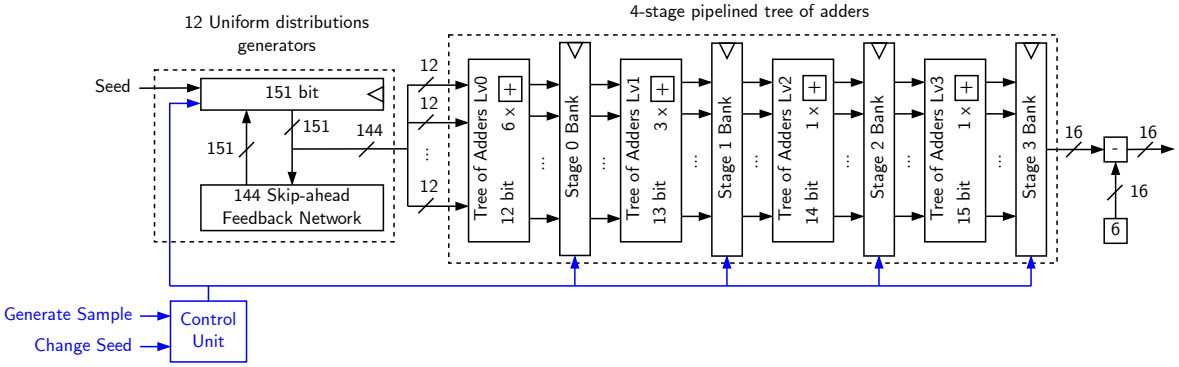


Figura 6.3: Diagrama del GRNG implementado. Las señales de control se muestran en azul, las de datos en negro.

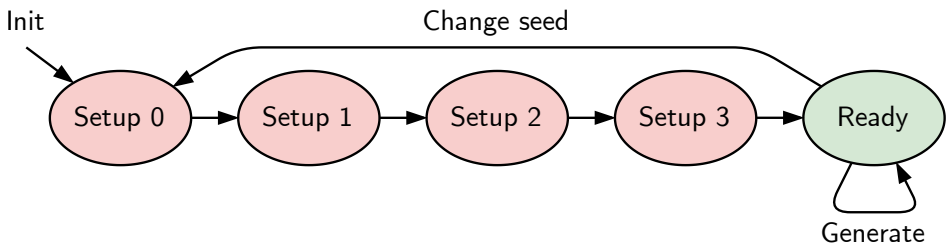


Figura 6.4: Diagrama de estados de la unidad de control del GRNG. El estado en el que se pueden generar muestras válidas esta marcado en verde, el resto en rojo.

6.2. Modificaciones del procesador RISC-V base

El GRNG diseñado se ha integrado en la CPU original como una nueva UF en la etapa *Execute*. Se ha modificado la etapa de *Decode* para añadir 2 nuevas instrucciones y el selector de resultado de la etapa *Execute* para poder utilizar la nueva UF. La Figura 6.5 muestra un diagrama de la ruta de datos

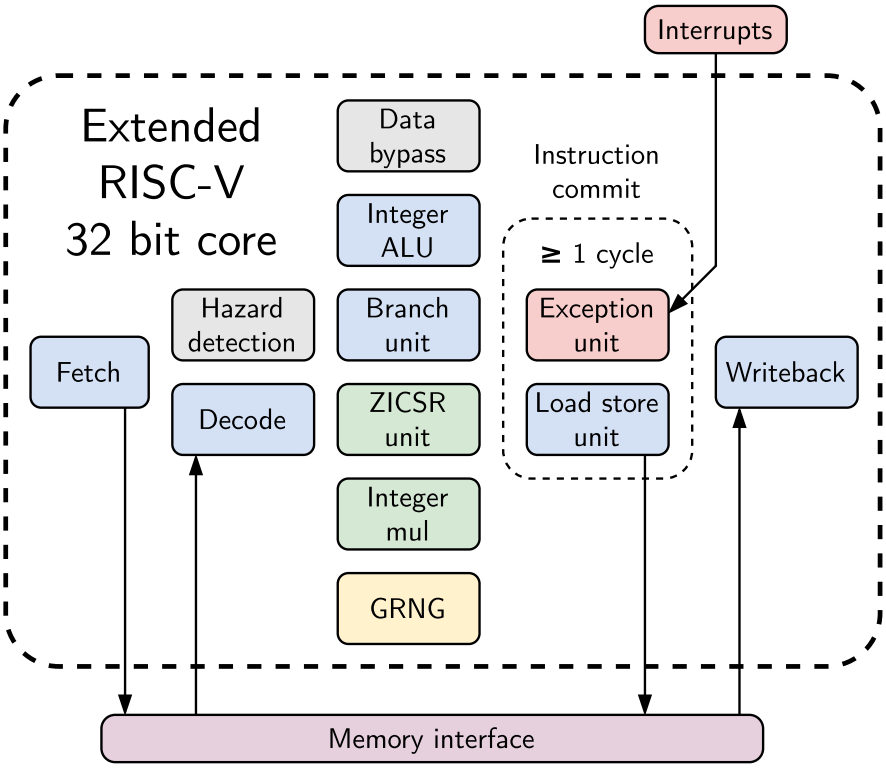


Figura 6.5: Ruta de datos del procesador RISC-V extendido con el GRNG, mostrado en amarillo. Los bloques relacionados con la arquitectura base RV32I se muestran en azul, los bloques para detectar riesgos de datos en gris, los bloques relacionados con el modo M en rojo, los bloques de extensiones estándar extra en verde y el interfaz con el bus de memoria en morado.

Otra opción posible habría sido añadirlo como un periférico que se accediera mediante entrada y salida mapeada en memoria (**M***emory* **M***apped* **I***nput* **O***utput*), pero esta fue descartada por los siguientes motivos. Implicaría que se accedería mediante instrucciones `load` y `store`, lo que reduciría el rendimiento ya que estas instrucciones no permiten consumidores a distancia 1. Además el acceso a memoria generalmente requiere circuitos mas complejos por lo que estas instrucciones pueden tener un coste mas elevado, esto no ocurre en la CPU RISC-V utilizada, cuyo acceso a memoria es lo más simple posible, pero esto si que podría ocurrir en otras CPU y afectar negativamente al rendimiento. Otro factor en contra de esta aproximación es que en el futuro se quiere crear una UF mas compleja que utilice el GRNG. El código VHDL del procesador modificado aparece en la Figura 3.1 como componente 5.

6.3. Actualización del compilador gcc con nuevas instrucciones

Para poder utilizar el GRNG se han añadido 2 instrucciones nuevas al repertorio RISC-V. El objetivo de estas nuevas instrucciones es acelerar el motor de inferencia que se ha desarrollado, por lo que se han de poder utilizar desde código de alto nivel en C. Para poder lograrlo, se ha actualizado el compilador `gcc` para RISC-V, lo que se ha podido hacer gracias a que es de código abierto. El compilador actualizado aparece en la Figura 3.1 como componente 4. A continuación se describen las instrucciones añadidas:

- `setseed rs1`. Cambia la semilla por el valor del registro `rs1`.

- `genum rd`. Genera una muestra de $\mathcal{N}(0,1)$ y la guarda en los 16 bits mas altos del registro `rd`. Eso implica que la muestra está codificada en coma fija en escala 2^{12+16}.

Para poder actualizar el compilador hay que definir la codificación de las instrucciones siguiendo las directrices de RISC-V. Además hay que definir una máscara junto con un valor para que `gcc` pueda comprobar la codificación mediante una operación `and`. Para ambas instrucciones se ha utilizado la codificación de instrucciones tipo R de RISC-V y se muestran en las Tablas 6.1 y 6.2.

Tabla 6.1: Codificación, máscara y validación de la instrucción `setseed` separada en los campos de instrucción RISC-V tipo R.

	31 25	24 20	19 15	14 12	11 7	6 0
	funct7	rs2	rs1	funct3	rd	opcode
`setseed rs1`	0000000	00000	xxxxx	000	00000	0001011
Mask	1111111	11111	00000	111	11111	1111111
Match	0000000	00000	00000	000	00000	0001011

Tabla 6.2: Codificación, máscara y validación de la instrucción `genum` separada en los campos de instrucción RISC-V tipo R.

	31 25	24 20	19 15	14 12	11 7	6 0
	funct7	rs2	rs1	funct3	rd	opcode
`genum rd`	0000000	00000	00000	001	xxxxx	0001011
Mask	1111111	11111	11111	111	00000	1111111
Match	0000000	00000	00000	001	00000	0001011

6.4. Análisis de resultados

Utilizando la extensión desarrollada no se degrada la precisión ni las métricas de incertidumbre con respecto a los resultados obtenidos con el motor de inferencia en C, ya que el algoritmo de muestreo es el mismo. La Tabla 6.3 muestra el *speedup* obtenido con todas las optimizaciones desarrolladas en este trabajo para todas las arquitecturas de modelos. Utilizando la extensión se obtiene el mejor rendimiento, llegando hasta un 7.65 en el mejor de los casos.

Tabla 6.3: *Speedups* obtenidos con todas las optmizaciones desarrolladas en este trabajo para todas las arquitecturas de modelos.

Modelo	**Speedup**		
	Bernoulli	*Uniforme*	*Extensión*
HYPER	5.18	4.95	6.37
LENET-5	4.28	3.87	5.01
B2N2	5.67	5.64	7.65

6.5. Análisis de coste

Para analizar el coste de añadir la extensión a la CPU base se ha utilizado la FPGA Xilinx ZCU104. Una FPGA es un componente que permite implementar en hardware real circuitos lógicos definidos en un lenguaje de alto nivel como VHDL. Las FPGA estan compuestas por distintos tipos de bloques, las herramientas de síntesis transforman el diseño a estos bloques y configuran la red de interconexión de los mismos. Algunos tipos de bloques son LUT, registros (**F***lip*-**F***lops*), memorias RAM (BRAM) o bloques para procesado de señal (**D***igital* **S***ignal* **P***rocessor*), que se pueden utilizar para realizar operaciones matemáticas complejas como la multiplicación. Utilizando una herramienta de síntesis se han obtenido los costes descritos en la Tabla 6.4.

Tabla 6.4: Utilización de recursos de la FPGA Xilinx ZCU104 por la CPU base y la extensión.

Tipo	**Recursos Utilizados**		**Utilización FPGA %**
	CPU Base	*Extensión GRNG*	
LUT	2435	240	1.16
FF	1922	320	0.49
BRAM	16	0	5.13
DSP	12	0	0.69

Añadir la extensión a la CPU base implica un incremento del 9.86 y del 16.65 % en LUTs y FFs. Hay que tener en cuenta que este coste además del GRNG incluye las modificaciones a la ruta de datos del procesador. Aun así el coste total de la CPU es muy bajo, no llegando

a utilizar ni un 10 % de los bloques de la FPGA. Además al haber segmentado el GRNG este no tiene ningun efecto negativo en la frecuencia del reloj del diseño original, 100 MHz.

La herramienta de síntesis también calcula estimaciones del consumo energético del diseño implementado, dividiéndolo en estático y dinámico. La Tabla 6.5 muestra esta estimación. El consumo energético aumenta solamente en un 0.65 %. Por lo tanto, la extensión consigue una reducción en del consumo casi idéntica a la ganancia en rendimiento.

Tabla 6.5: Estimación del consumo energético en la FPGA Xilinx ZCU104 de la CPU base y extendida.

Tipo	**Consumo Energético (W)**		
	CPU Base	*Extensión GRNG*	Total
Dinámico	0.023	0.004	0.027
Estático	0.593	0.000	0.593

Capítulo 7

Conclusiones

Este trabajo estudia la inferencia de BNN en dispositivos IoT de bajo consumo, utilizando como ejemplo un procesador RISC-V. Se han desarrollado herramientas de código abierto para transformar modelos BNN en punto flotante de TensorFlow a código C para inferencia utilizando solo precisión entera. Además, analiza diferentes métodos de optimización para el algoritmo de muestreo de pesos del algoritmo de inferencia, ya que consume la mayor parte del tiempo de ejecución.

Para acelerar la inferencia, este trabajo estudia el impacto en las métricas de incertidumbre y el coste de su implementación en software de una optimización propuesta en otro trabajo. Además este trabajo propone una nueva versión de dicha optimización que consiste en muestrear una distribución Uniforme. Esta optimización logra, en promedio, un *speedup* de 5 en varios modelos representativos de BNN, conservando las métricas de incertidumbre en todos ellos.

Para mejorar estos resultados, se ha desarrollado una extensión para RISC-V que permite muestrear una distribución Gaussiana usando solo una instrucción, implementada en un procesador de 32 bits. Las muestras se generan utilizando un GRNG basado en el TCL de bajo coste. La extensión acelera la inferencia hasta 7.65 veces, con reducciones similares en el consumo de energía y sin degradación significativa en la precisión o las métricas de incertidumbre. Se ha implementado el diseño en una FPGA Xilinx ZCU104. Su coste es solo de 240 LUTs y 320 Flip-Flops, generando un aumento del 0.65 % en el consumo de energía, sin afectar la frecuencia del reloj del sistema.

Los resultados de este trabajo han sido presentados en la conferencia IEEE NANO 2024. También han sido presentados en la Jornada de Jóvenes Investigadores 2024 del I3A como presentación oral.

Como trabajo futuro se podría seguir iterando sobre la UF desarrollada permitiendo no solo generar muestras sino realizar toda la operación MAC requerida en la inferencia, lo que aumentaría aun mas el rendimiento. También se podría estudiar la portabilidad de la extensión desarrollada incluyéndola en otras CPU RISC-V abiertas. Por el lado del software,

se podría añadir al motor desarrollado un marco de cuantización mas complejo, permitiendo reducir aun mas el tamaño de los modelos.

Capítulo 8

Bibliografía

[1] Goldman Sachs. Artificial intelligence (AI) market interest growth 2015 to 2023, by share of companies [graph]. https://www.statista.com/statistics/1424672/ai-market-interest-worldwide/, August 2023.

[2] Statista. Market size and revenue comparison for artificial intelligence worldwide from 2020 to 2030 (in billion U.S. dollars) [Graph]. https://www.statista.com/statistics/941835/artificial-intelligence-market-size-revenue-comparisons/, May 2024.

[3] La Ley de Inteligencia Artificial de la UE. https://artificialintelligenceact.eu/es/.

[4] European Processor Initiative. https://www.european-processor-initiative.eu/.

[5] Shvetank Prakash, Matthew Stewart, Colby Banbury, Mark Mazumder, Pete Warden, Brian Plancher, and Vijay Janapa Reddi. Is tinyml sustainable? *Commun. ACM*, 66(11):68–77, oct 2023. https://doi.org/10.1145/3608473.

[6] Samuel Pérez Pedrajas, Javier Resano Ezcaray, and Darío Suárez Gracia. Implementación de un procesador RISC-V con soporte para un sistema operativo de tiempo real. 2022.

[7] Samuel Pérez Pedrajas, Javier Resano Ezcaray, and Darío Suárez Gracia. GitHub. BNN_RISC-V. https://github.com/Samulix20/BNN_RISC-V, 2024.

[8] RISC-V International. RISC-V: The Open Standard RISC Instruction Set Architecture. https://riscv.org/.

[9] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20240411*. RISC-V Foundation, April 2024. https://riscv.org/technical/specifications/.

[10] Editors Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged ISA, Document Version 20240411*. RISC-V Foundation, April 2024. https://riscv.org/technical/specifications/.

[11] Codasip. Architect your ambition with RISC-V Custom Compute. https://codasip.com/.

[12] Synopsys. RISC-V. https://www.synopsys.com/risc-v.html#p-design.

[13] Víctor Soria-Pardos, Max Doblas, Guillem López–Paradís, Gerard Candón, Narcís Rodas, Xavier Carril, Pau Fontova–Musté, Neiel Leyva, Santiago Marco-Sola, and Miquel Moretó. Sargantana: A 1 ghz+ in-order risc-v processor with simd vector extensions in 22nm fd-soi. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, pages 254–261, 2022.

[14] Antonio Pullini, Davide Rossi, Igor Loi, Giuseppe Tagliavini, and Luca Benini. Mr.wolf: An energy-precision scalable parallel ultra low power soc for iot edge processing. *IEEE Journal of Solid-State Circuits*, 54(7):1970–1981, 2019.

[15] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, Apr 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[17] Leiyu Chen, Shaobo Li, Qiang Bai, Jing Yang, Sanlong Jiang, and Yanming Miao. Review of image classification algorithms based on convolutional neural networks. *Remote Sensing*, 13(22), 2021.

[18] Adrián Alcolea and Javier Resano. Bayesian neural networks to analyze hyperspectral datasets using uncertainty metrics. *IEEE Transactions on Geoscience and Remote Sensing*, 60:1–10, 2022.

[19] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks, 2015.

[20] S Kullback and R A Leibler. On information and sufficiency. *Ann. Math. Stat.*, 22(1):79–86, March 1951.

[21] Umang Bhatt, Yunfeng Zhang, Javier Antorán, Q. Vera Liao, Prasanna Sattigeri, Riccardo Fogliato, Gabrielle Gauthier Melançon, Ranganath Krishnan, Jason Stanley, Omesh Tickoo, Lama Nachman, Rumi Chunara, Adrian Weller, and Alice Xiang. Uncertainty as a form of transparency: Measuring, communicating, and using uncertainty. *CoRR*, abs/2011.07586, 2020. `https://arxiv.org/abs/2011.07586`.

[22] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Ai and ml accelerator survey and trends. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–10, 2022.

[23] Ruizhe Cai, Ao Ren, Ning Liu, Caiwen Ding, Luhao Wang, Xuehai Qian, Massoud Pedram, and Yanzhi Wang. VIBNN: hardware acceleration of bayesian neural networks. *CoRR*, abs/1802.00822, 2018. `http://arxiv.org/abs/1802.00822`.

[24] Hiromitsu Awano and Masanori Hashimoto. Bynqnet: Bayesian neural network with quadratic activations for sampling-free uncertainty estimation on fpga. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1402–1407, 2020.

[25] Hiromitsu Awano and Masanori Hashimoto. B2n2: Resource efficient bayesian neural network accelerator using bernoulli sampler on fpga. *Integration*, 89:1–8, 2023. `https://www.sciencedirect.com/science/article/pii/S0167926022001523`.

[26] C. S. Wallace. Fast pseudorandom generators for normal and exponential variates. *ACM Trans. Math. Softw.*, 22(1):119–127, mar 1996. `https://doi.org/10.1145/225545.225554`.

[27] Jamshaid Sarwar Malik and Ahmed Hemani. Gaussian random number generation: A survey on hardware architectures. *ACM Comput. Surv.*, 49(3), nov 2016. https://doi.org/10.1145/2980052.

[28] Jamshaid Sarwar Malik, Ahmed Hemani, Jameel Nawaz Malik, Ben Silmane, and Nasirud Din Gohar. Revisiting central limit theorem: Accurate gaussian random number generation in vlsi. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(5):842–855, 2015.

[29] Yuki Hirayama, Tetsuya Asai, Masato Motomura, and Shinya Takamaeda. A hardware-efficient weight sampling circuit for bayesian neural networks. *International Journal of Networking and Computing*, 10(2):84–93, 2020. http://www.ijnc.org/index.php/ijnc/article/view/222.

[30] Google Brain Team. TensorFlow Probability. `https://www.tensorflow.org/probability`.

[31] Ranganath Krishnan, Pi Esposito, and Mahesh Subedar. Bayesian-torch: Bayesian neural network layers for uncertainty estimation. `https://github.com/IntelLabs/bayesian-torch`.

[32] Google Brain Team. TensorFlow Lite — ML for Mobile and Edge Devices. `https://www.tensorflow.org/lite`.

[33] Jun-Liang Lin, Ranganath Krishnan, Keyur Ruganathbhai Ranipa, Mahesh Subedar, Vrushabh Sanghavi, Meena Arunachalam, Omesh Tickoo, Ravishankar Iyer, and Mahmut Taylan Kandemir. Quantization for bayesian deep learning: Low-precision characterization and robustness. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 180–192. IEEE, 2023.

[34] Linus Torvalds, Junio Hamano, and Software Freedom Conservancy. Git. `https://www.git-scm.com/`.

[35] GitHub. `https://github.com/`.

[36] Samuel Pérez Pedrajas, Javier Resano Ezcaray, and Darío Suárez Gracia. GitHub. riscv-vhdl. `https://github.com/Samulix20/riscv-vhdl`, 2022.

[37] Adrián Alcolea and Javier Resano. GitHub. BNN_for_hyperspectral_datasets_analysis. `https://github.com/universidad-zaragoza/BNN_for_hyperspectral_datasets_analysis`, 2022.

[38] Tristan Gingold. GHDL. `http://ghdl.free.fr/`.

[39] GTKWave. `https://gtkwave.sourceforge.net/`.

[40] RISC-V Collaboration. GitHub. riscv-gnu-toolchain. `https://github.com/riscv-collab/riscv-gnu-toolchain`.

[41] Eli Bendersky. Github. pyelftools. `https://github.com/eliben/pyelftools`.

[42] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[43] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[44] AMD Xilinx. Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. `https://www.xilinx.com/products/boards-and-kits/zcu104.html#overview`.

[45] AMD Xilinx. Vivado Design Suite. `https://www.xilinx.com/products/design-tools/vivado.html`.

[46] The Overleaf Team. Overleaf, Online LaTeX Editor. `https://es.overleaf.com/`.

[47] JGraph. draw.io. `https://app.diagrams.net/`.

[48] Inkscape Project. Inkscape. https://inkscape.org.

[49] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[50] Yann LeCun, Corinna Cortes, and Chris Burges. MNIST handwritten digit database. `http://yann.lecun.com/exdb/mnist/`.

[51] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[52] Clay S Turner. A fast binary logarithm algorithm [DSP tips & tricks]. *IEEE Signal Processing Magazine*, 27(5):124–140, 2010.

[53] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003. `https://www.jstatsoft.org/index.php/jss/article/view/v008i14`.

[54] Peter Alfke. Xilinx. efficient shift registers, lfsr counters, and long pseudorandom sequence generators, July 1996.

[55] Pong P Chu and Robert E Jones. Design techniques of fpga based random number generator. In *Military and Aerospace Applications of Programmable Devices and Technologies Conference*, volume 1, pages 28–30. Citeseer, 1999.

[56] Leonard Colavito and Dennis Silage. Efficient pga lfsr implementation whitens pseudorandom numbers. In *2009 International Conference on Reconfigurable Computing and FPGAs*, pages 308–313, 2009.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Resultados de incertidumbre del motor de inferencia

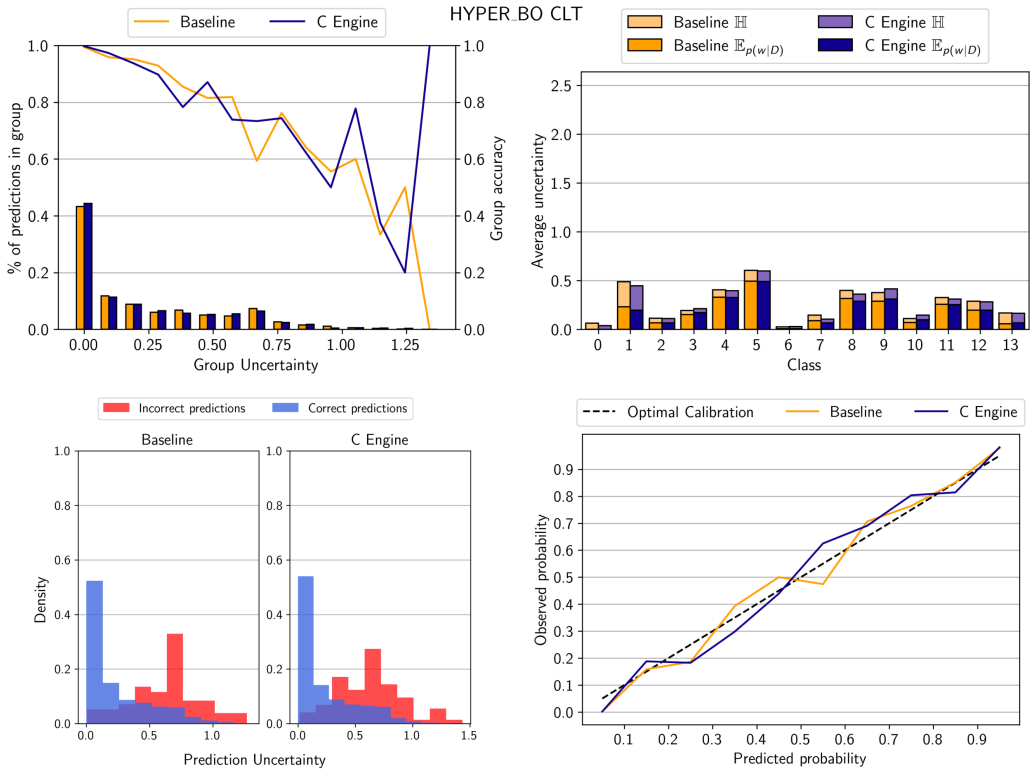


Figura A.1: Predicciones del conjunto de prueba de píxeles hiperespectrales BO.

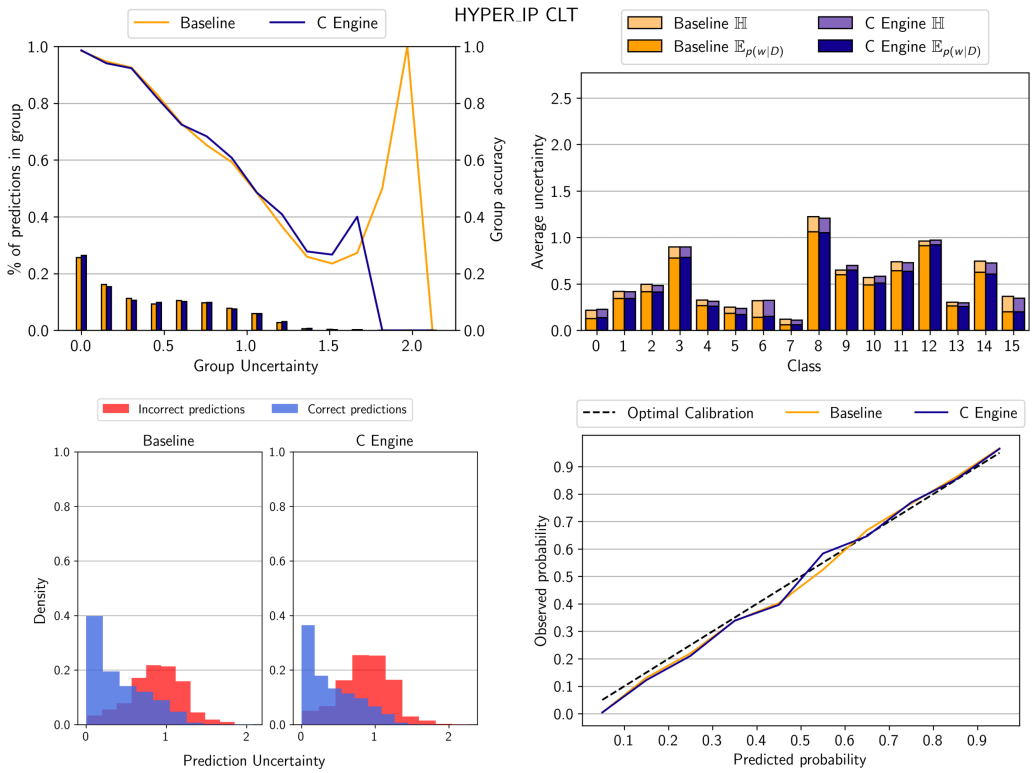


Figura A.2: Predicciones del conjunto de prueba de píxeles hiperespectrales IP.

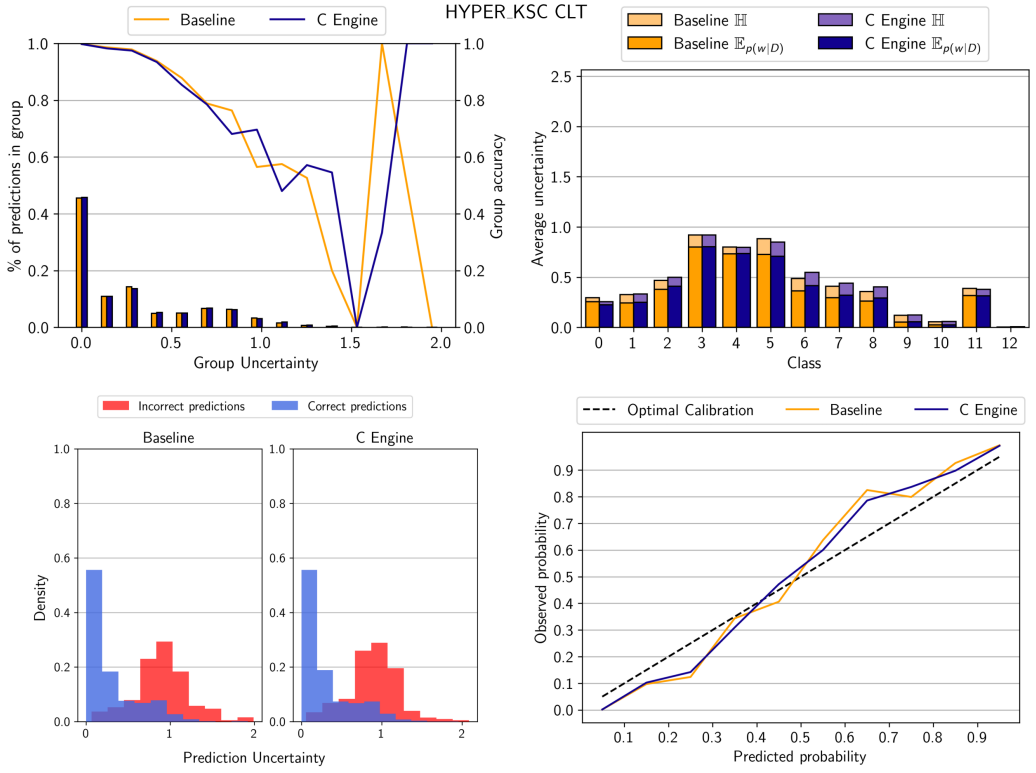


Figura A.3: Predicciones del conjunto de prueba de píxeles hiperespectrales KSC.

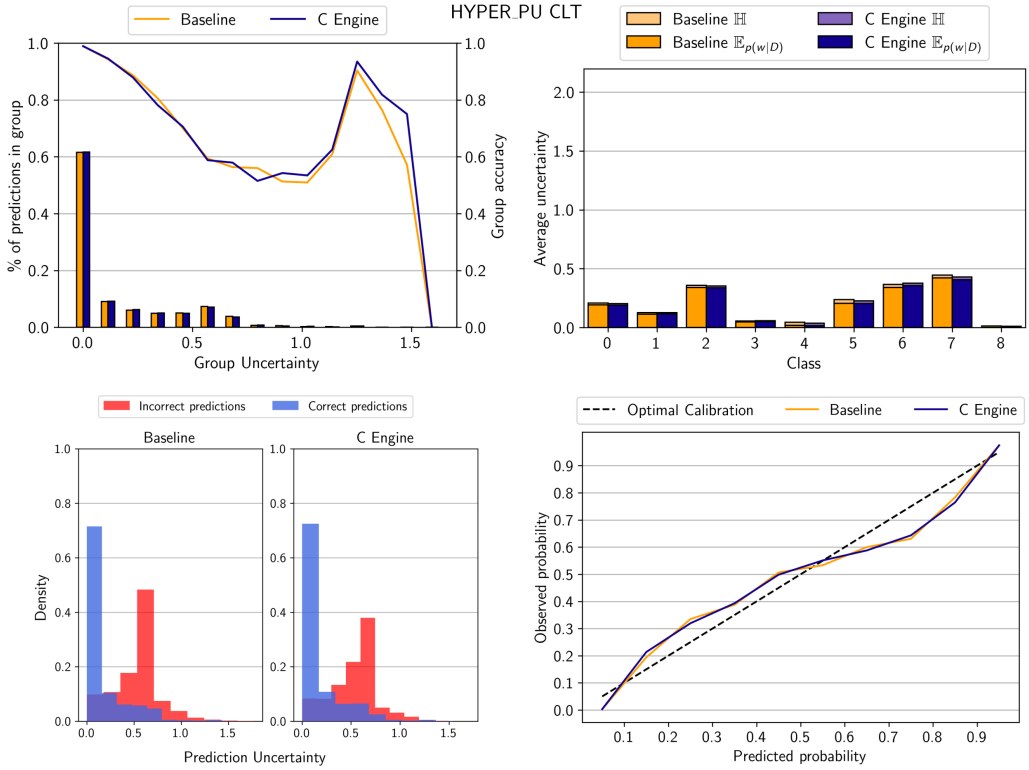


Figura A.4: Predicciones del conjunto de prueba de píxeles hiperespectrales PU.

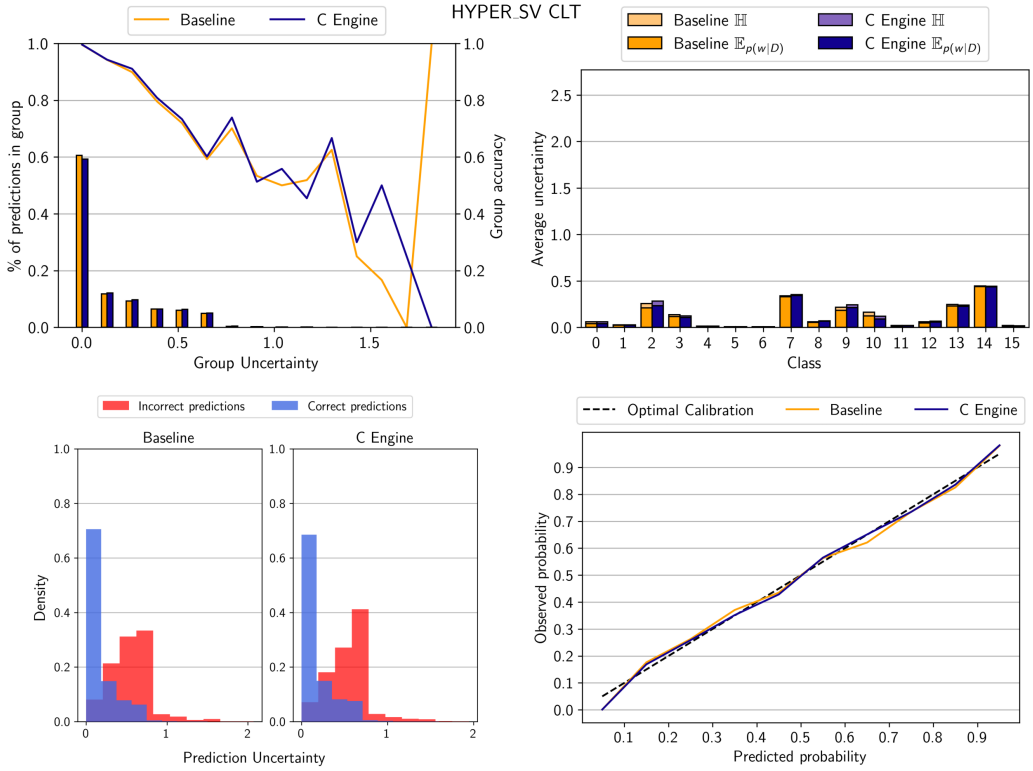


Figura A.5: Predicciones del conjunto de prueba de píxeles hiperespectrales SV.

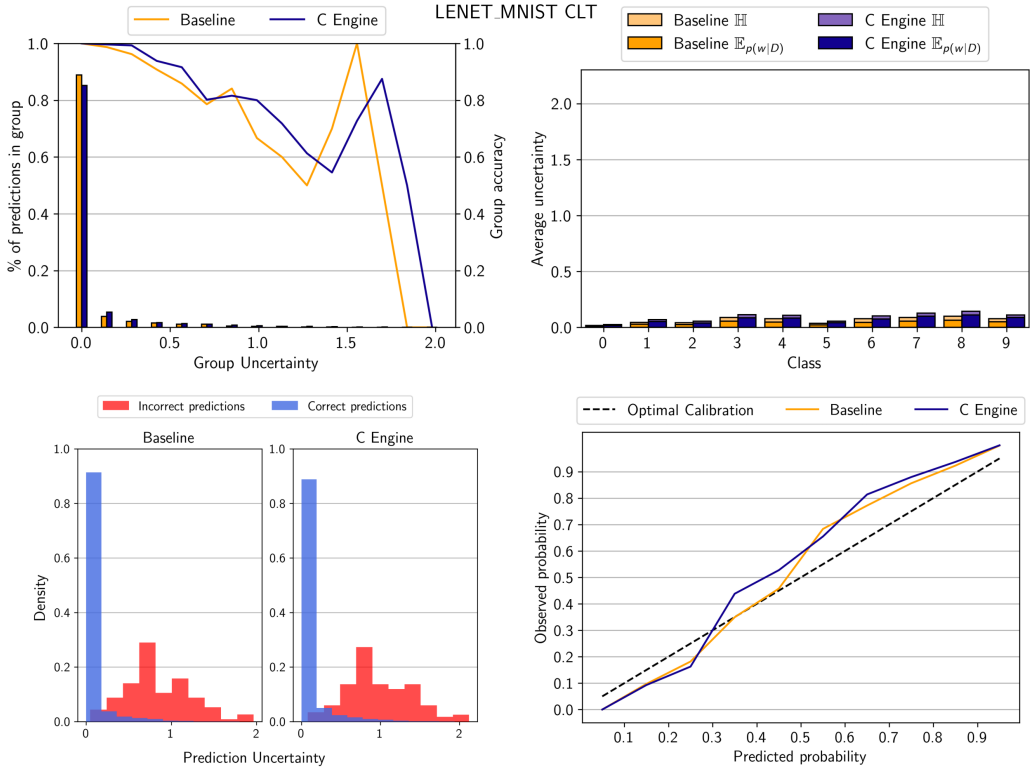


Figura A.6: Predicciones del modelo LENET con el conjunto de datos MNIST.

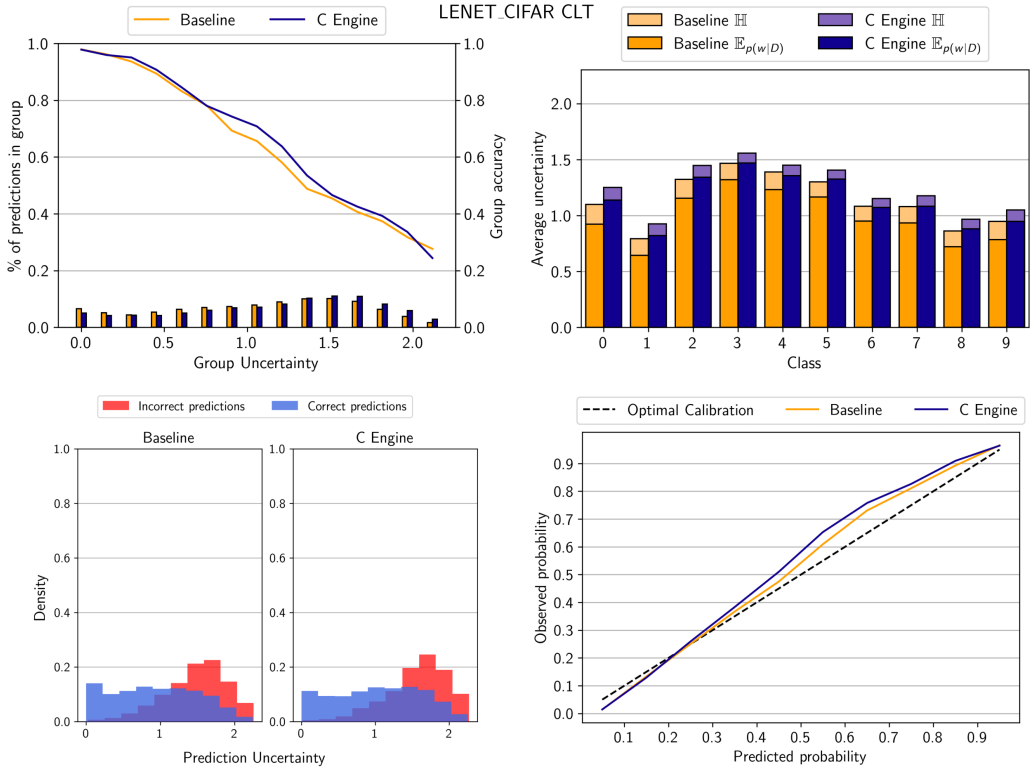


Figura A.7: Predicciones del modelo LENET con el conjunto de datos CIFAR-10.

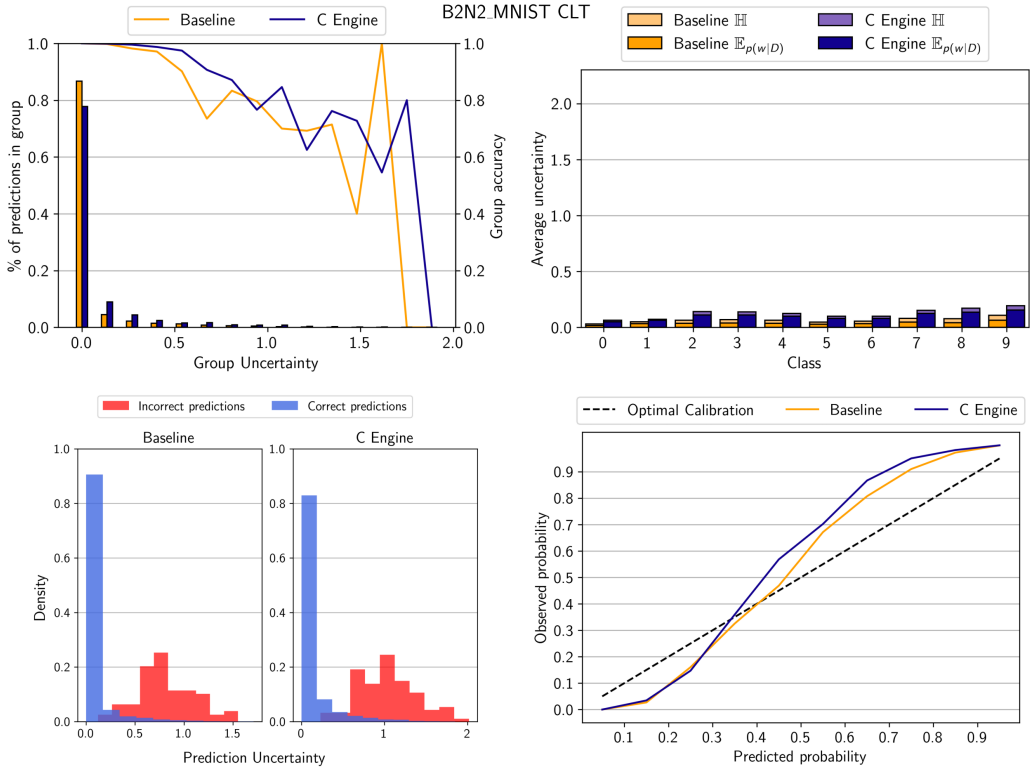


Figura A.8: Predicciones del modelo B2N2 con el conjunto de datos MNIST.

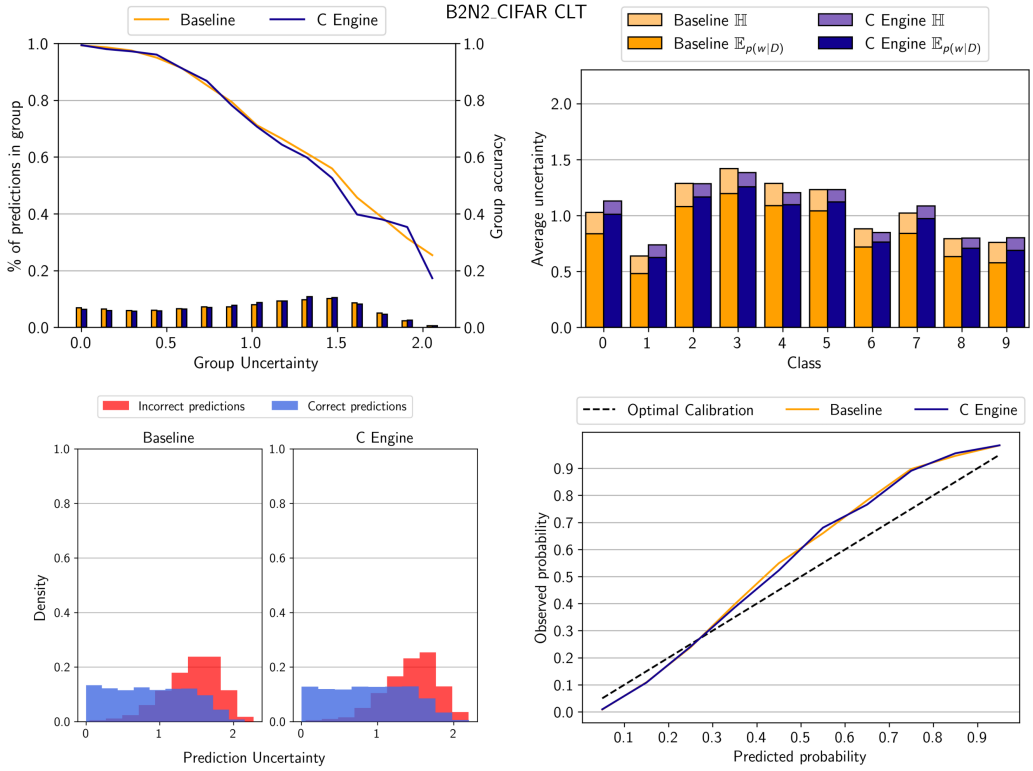


Figura A.9: Predicciones del modelo B2N2 con el conjunto de datos CIFAR-10.

Anexos B

Resultados de incertidumbre optimización Bernoulli

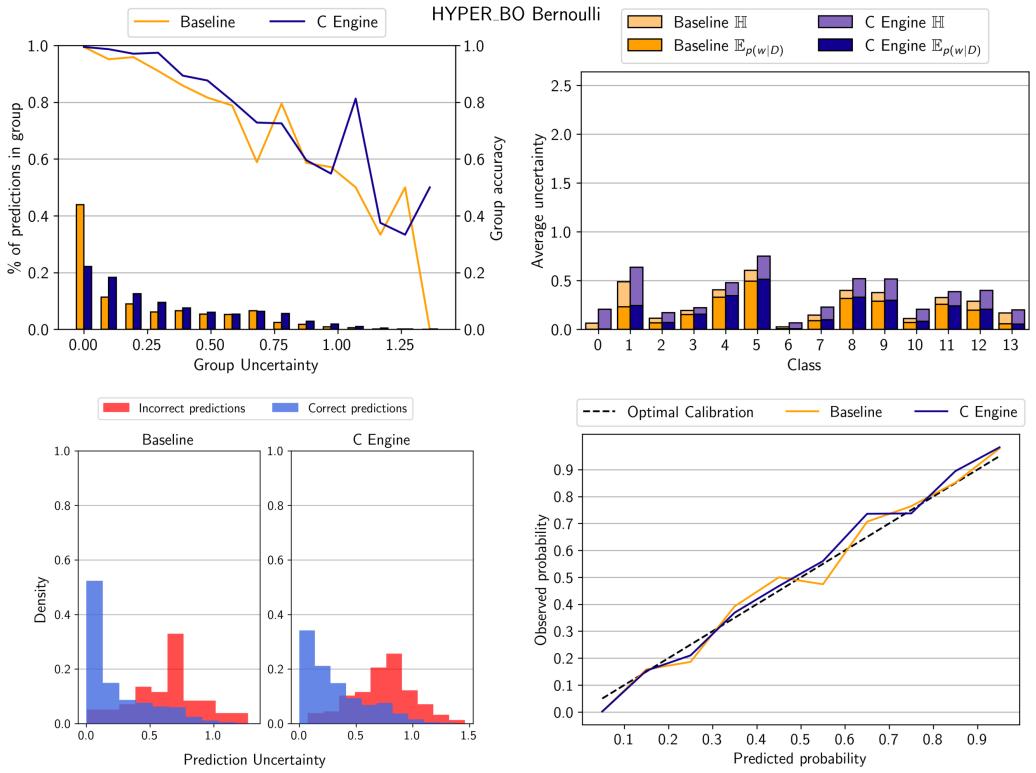


Figura B.1: Predicciones del conjunto de prueba de píxeles hiperespectrales BO.

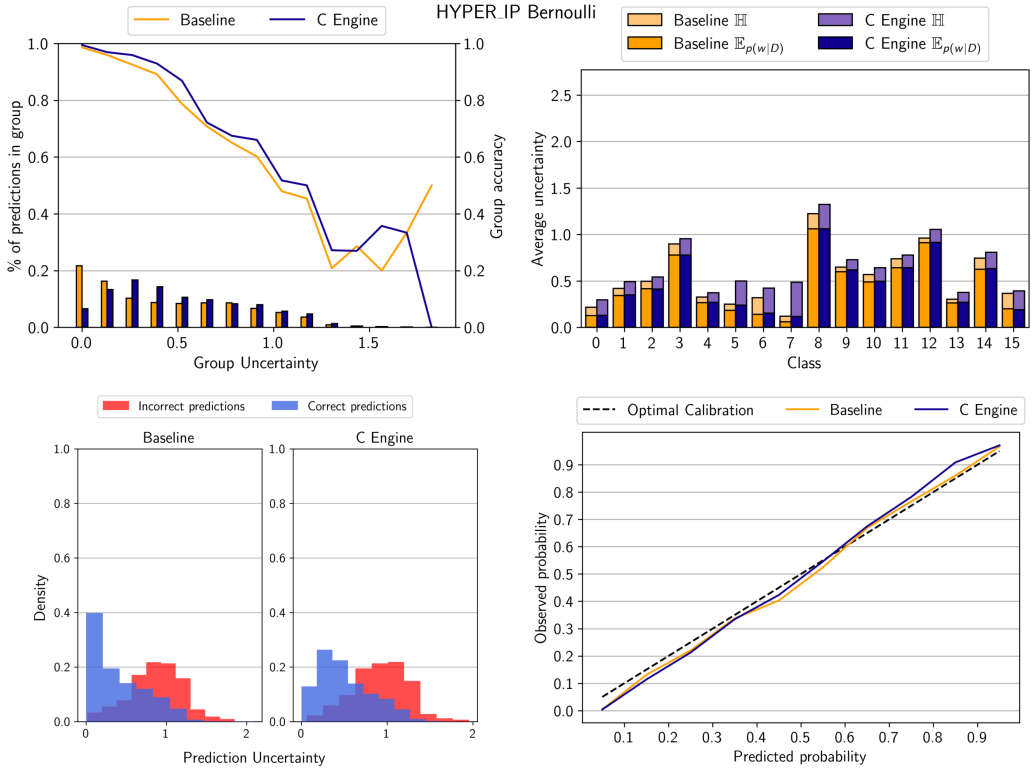


Figura B.2: Predicciones del conjunto de prueba de píxeles hiperespectrales IP.

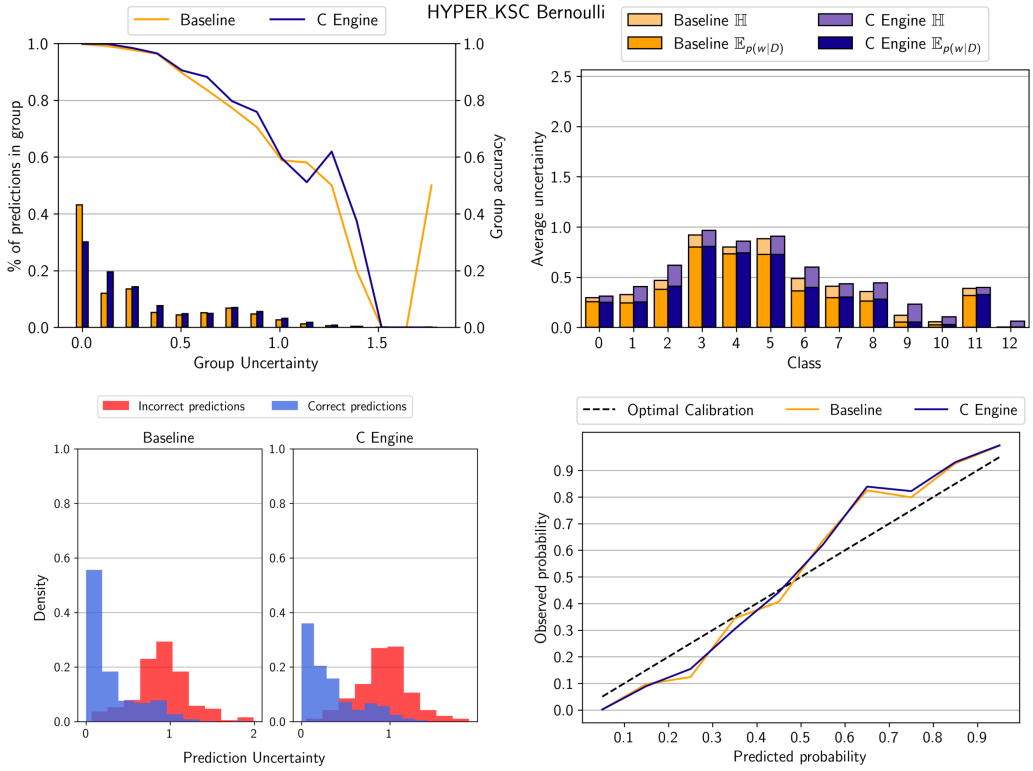


Figura B.3: Predicciones del conjunto de prueba de píxeles hiperespectrales KSC.

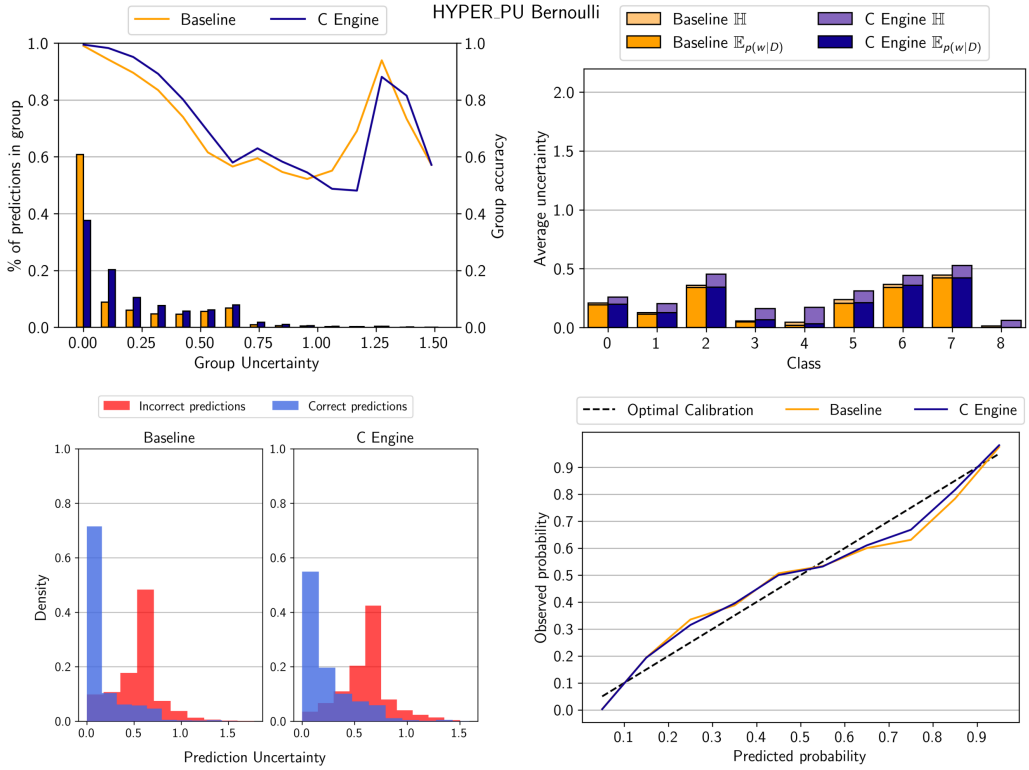


Figura B.4: Predicciones del conjunto de prueba de píxeles hiperespectrales PU.

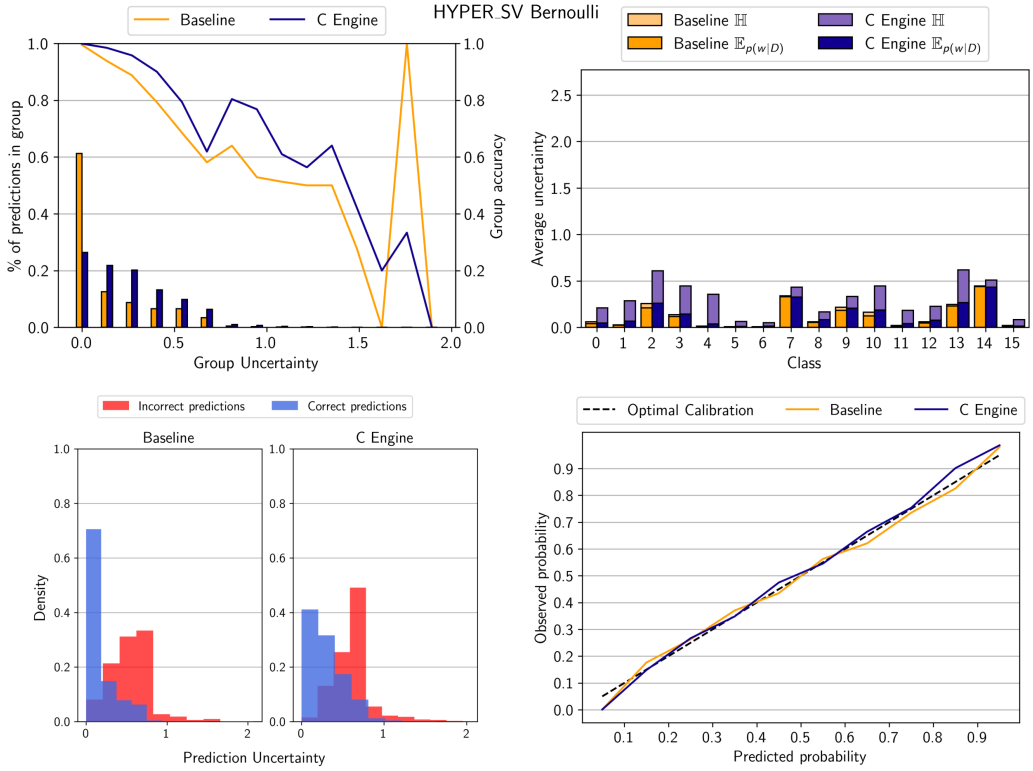


Figura B.5: Predicciones del conjunto de prueba de píxeles hiperespectrales SV.

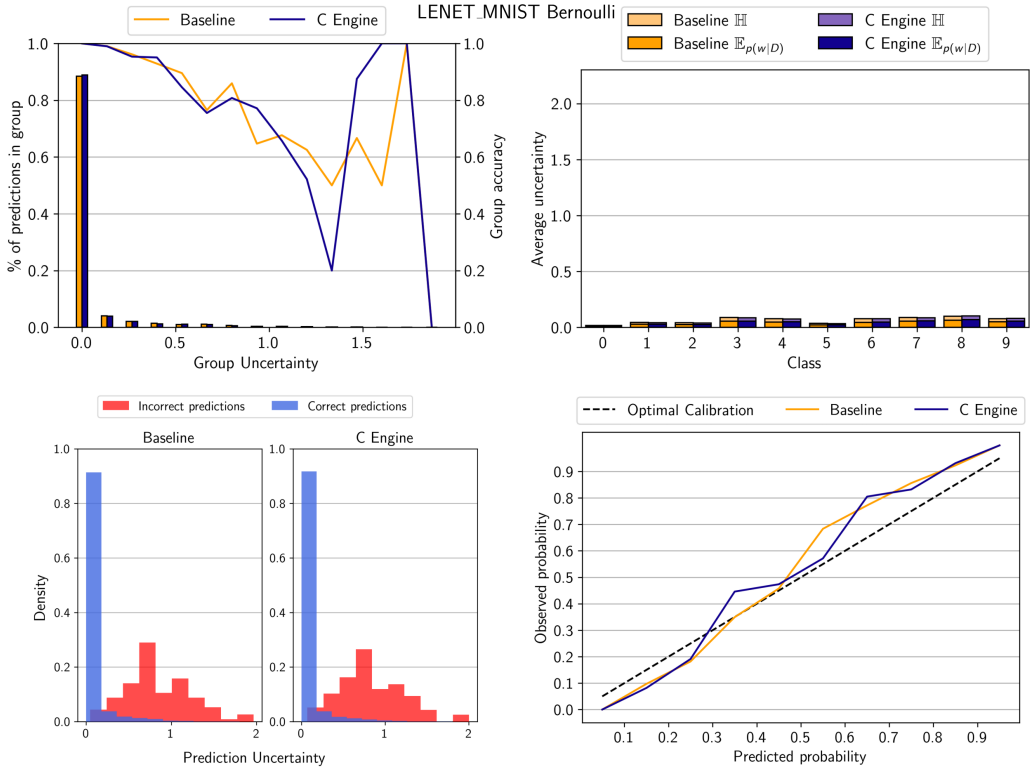


Figura B.6: Predicciones del modelo LENET con el conjunto de datos MNIST.

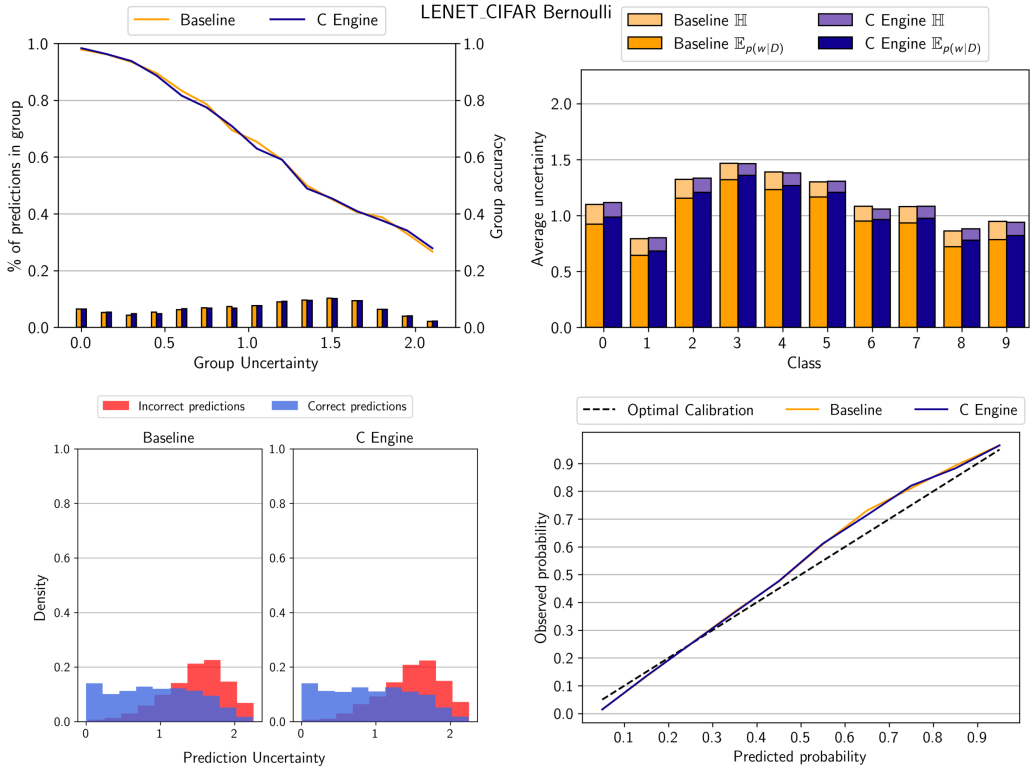


Figura B.7: Predicciones del modelo LENET con el conjunto de datos CIFAR-10.

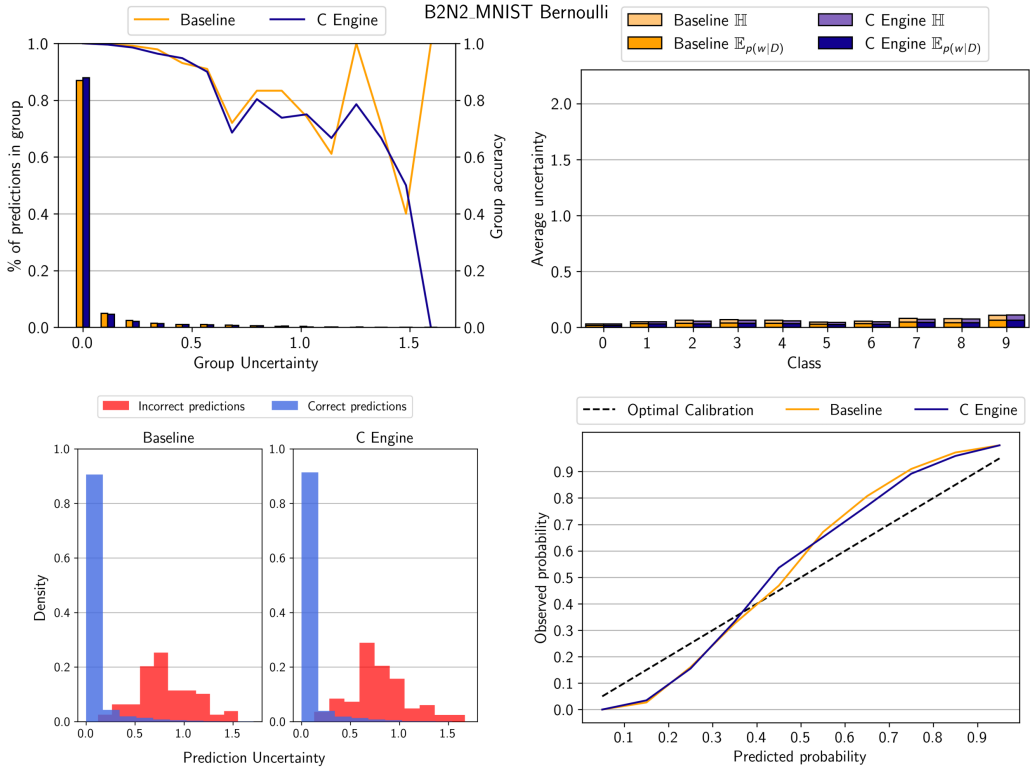


Figura B.8: Predicciones del modelo B2N2 con el conjunto de datos MNIST.

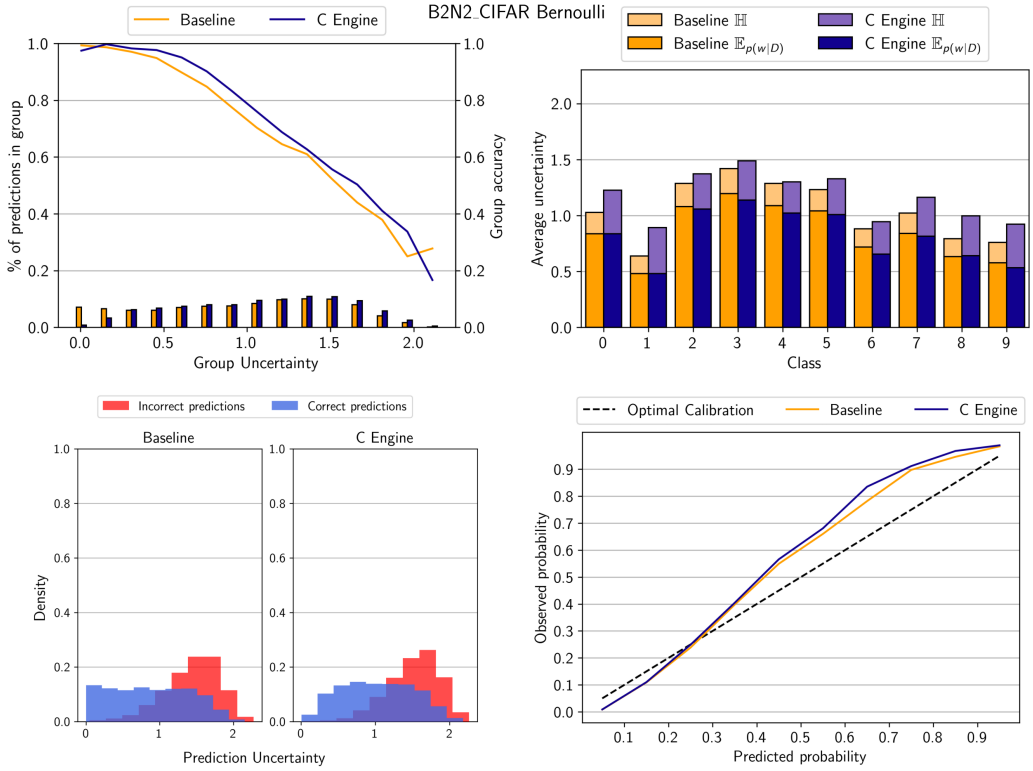


Figura B.9: Predicciones del modelo B2N2 con el conjunto de datos CIFAR-10.

Anexos C

Resultados de incertidumbre optimización Uniforme

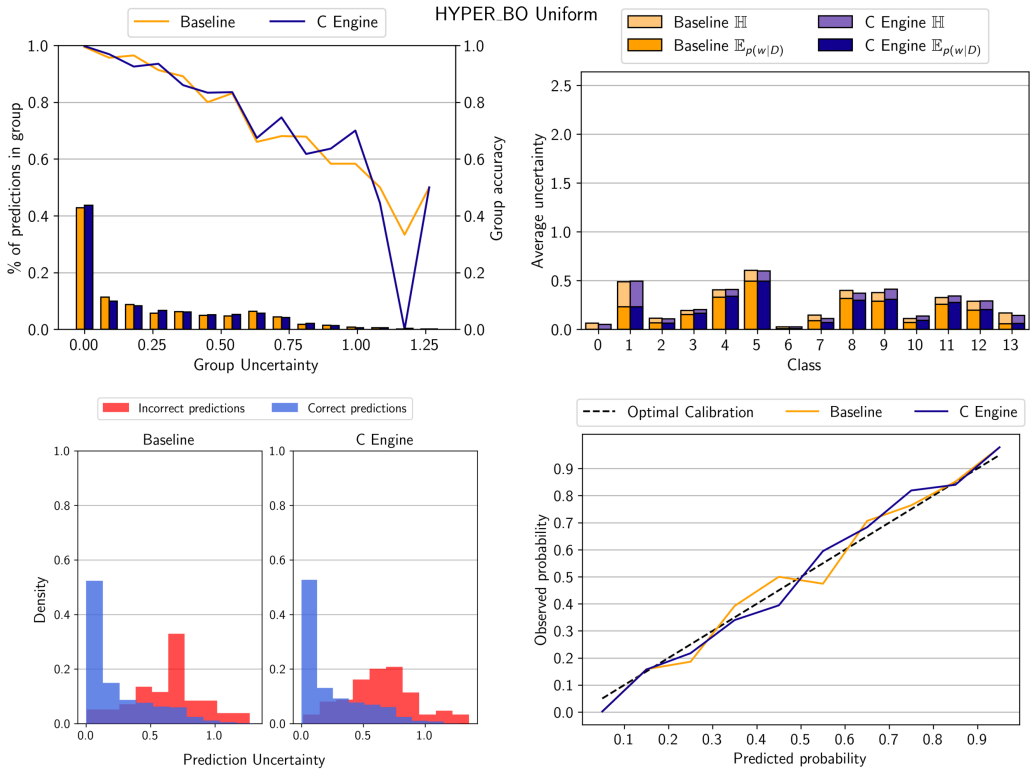


Figura C.1: Predicciones del conjunto de prueba de píxeles hiperespectrales BO.

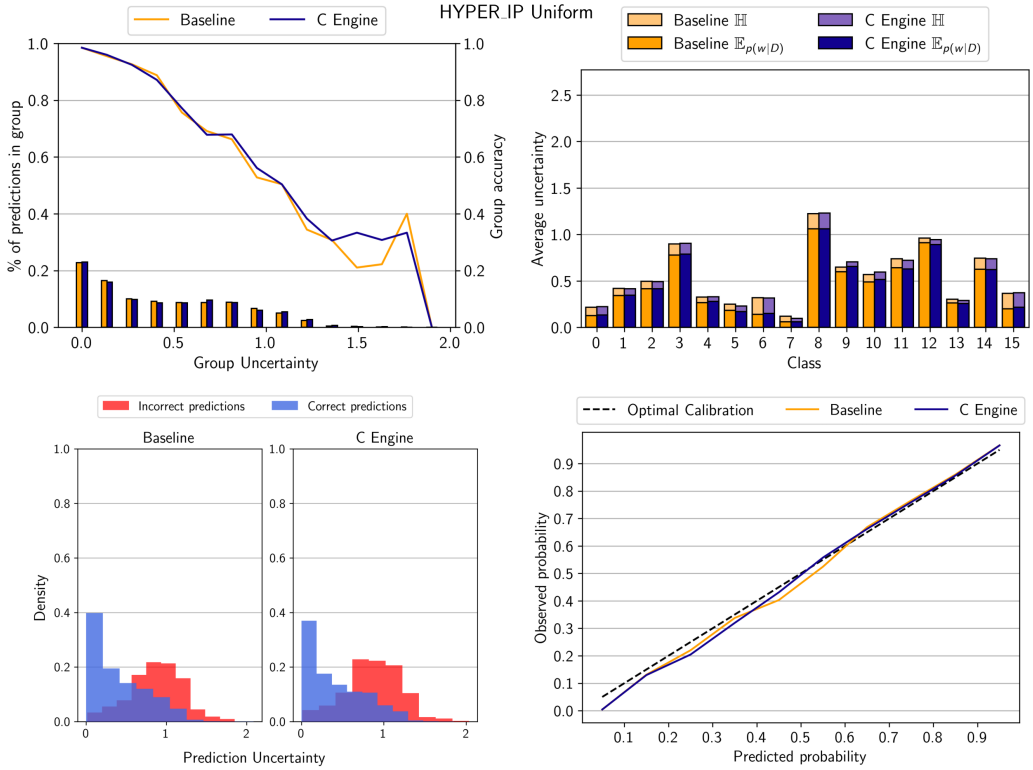


Figura C.2: Predicciones del conjunto de prueba de píxeles hiperespectrales IP.

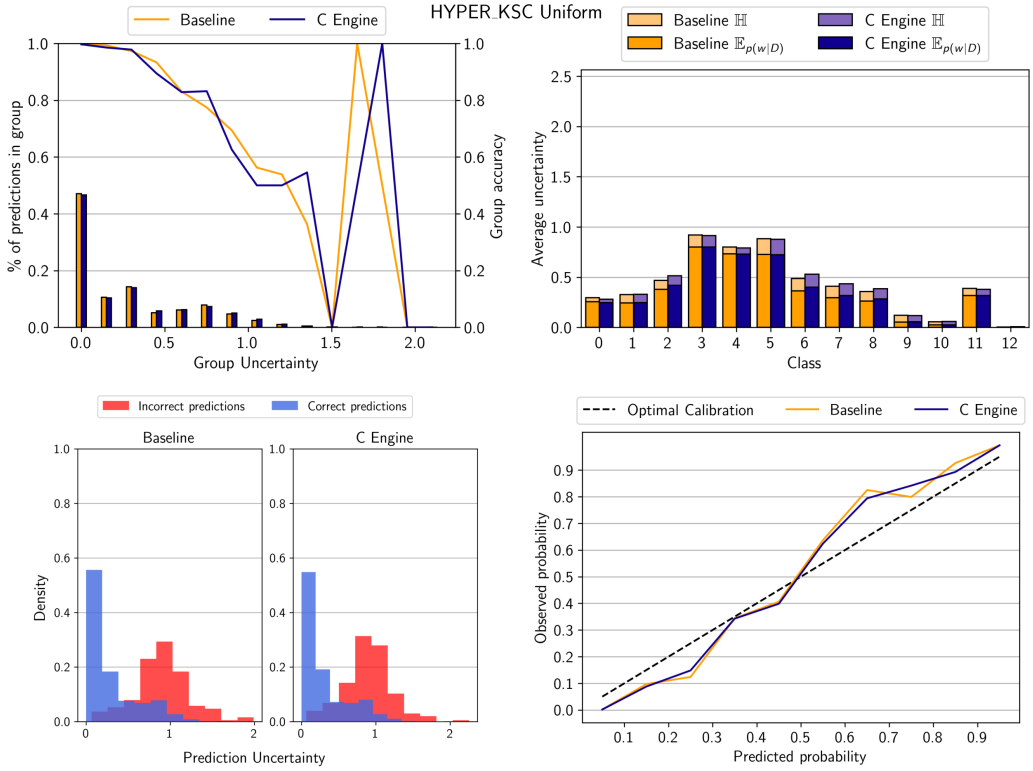


Figura C.3: Predicciones del conjunto de prueba de píxeles hiperespectrales KSC.

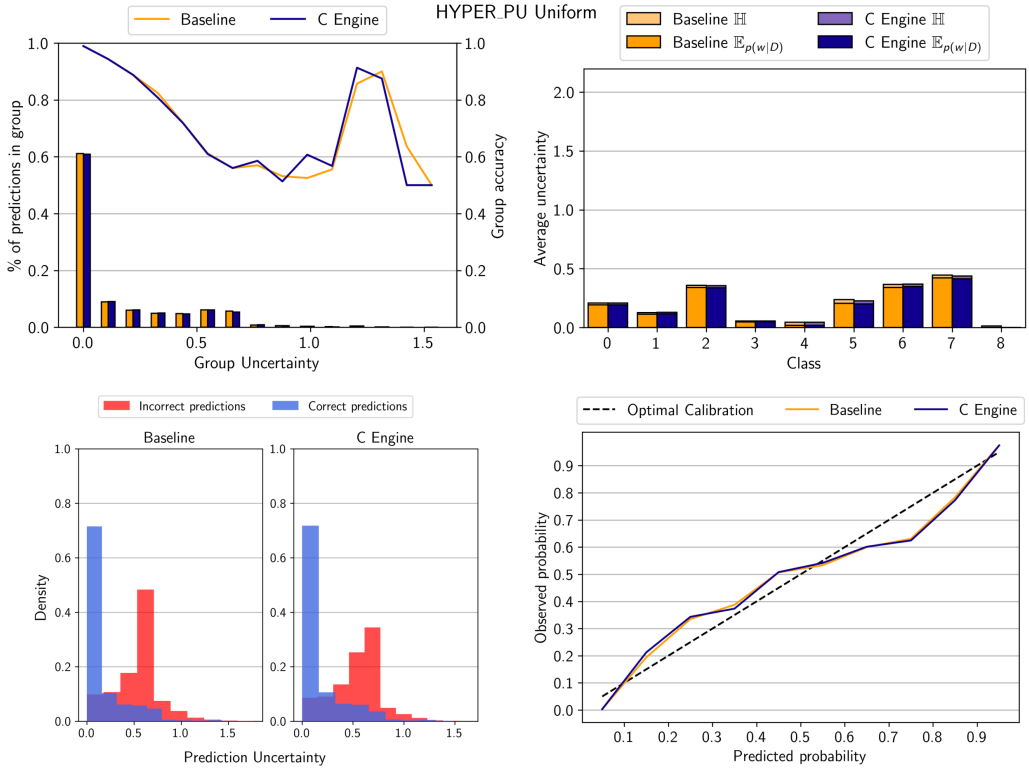


Figura C.4: Predicciones del conjunto de prueba de píxeles hiperespectrales PU.

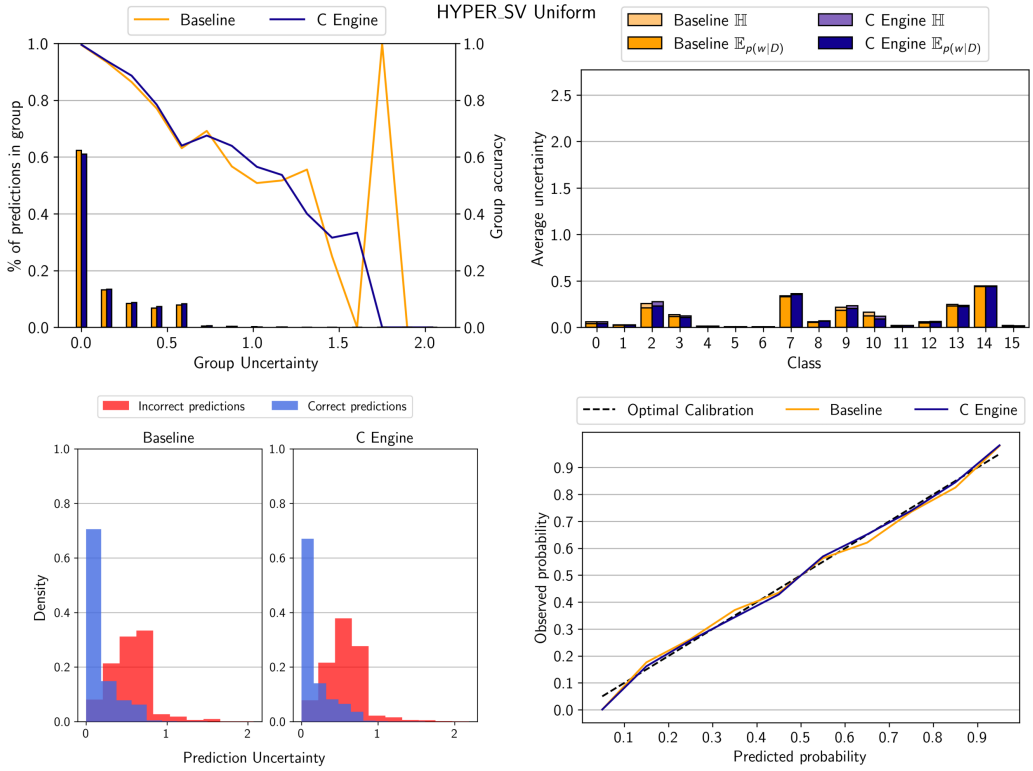


Figura C.5: Predicciones del conjunto de prueba de píxeles hiperespectrales SV.

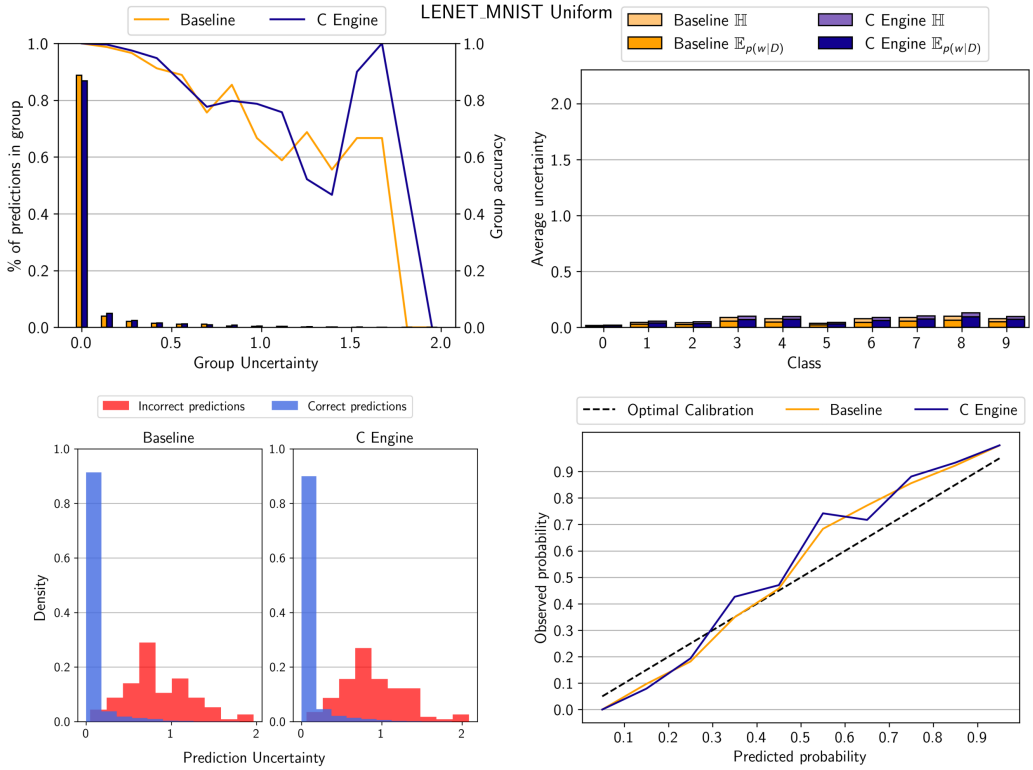


Figura C.6: Predicciones del modelo LENET con el conjunto de datos MNIST.

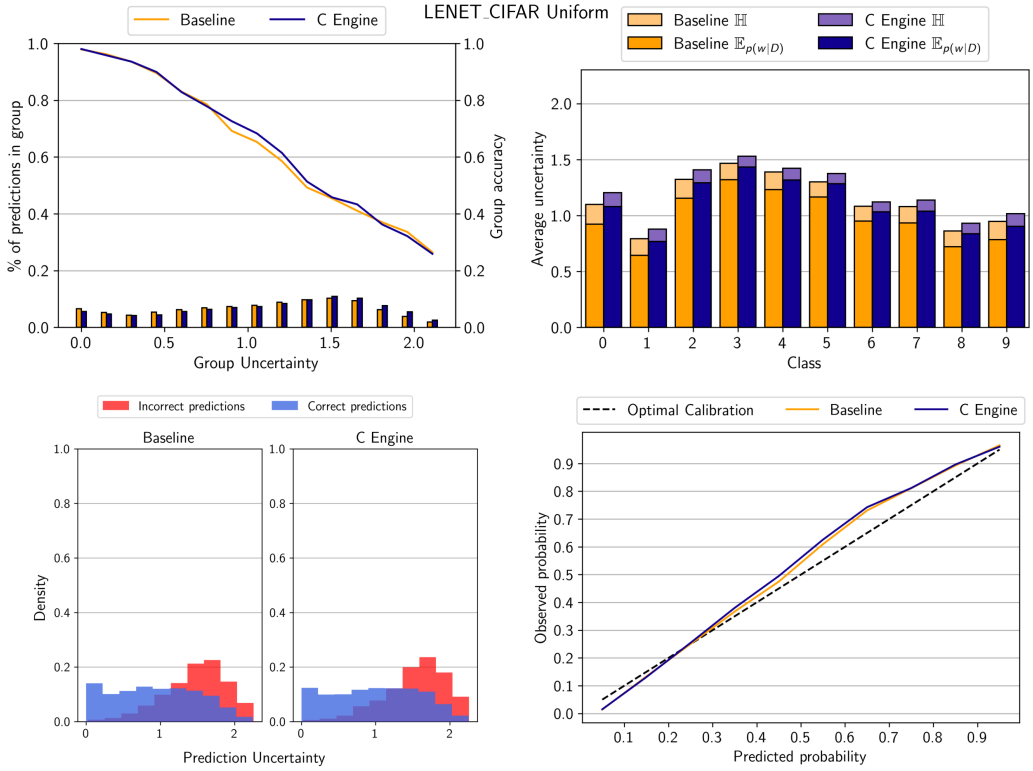


Figura C.7: Predicciones del modelo LENET con el conjunto de datos CIFAR-10.

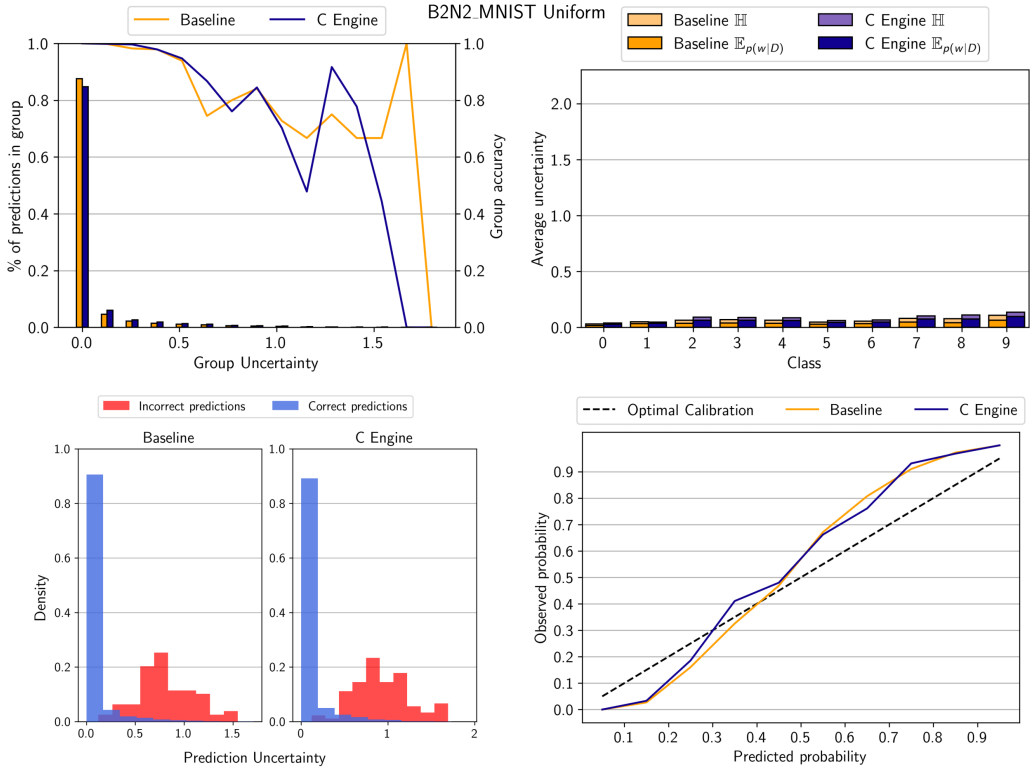


Figura C.8: Predicciones del modelo B2N2 con el conjunto de datos MNIST.

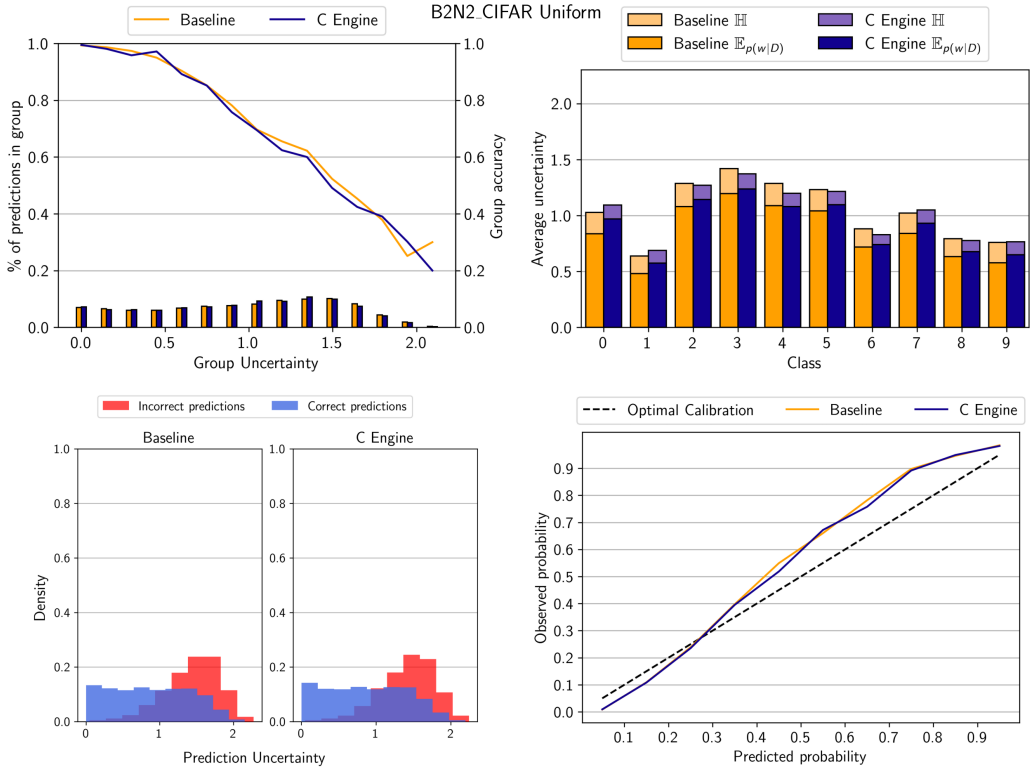


Figura C.9: Predicciones del modelo B2N2 con el conjunto de datos CIFAR-10.