



Escola Politécnica de Enxeñaría de Ferrol

UNIVERSIDADE DA CORUÑA



**Trabajo Fin de Grado
CURSO 2023/2024**

*Escalabilidad y eficiencia en Inteligencia Artificial
a través de algoritmos federados verdes*

Grado en Ingeniería en Tecnologías Industriales

AUTOR: ABEL PAMPÍN RODRÍGUEZ

TUTOR: ÓSCAR FONTENLA ROMERO

TFG N°: 2223

FECHA: SEPTIEMBRE DE 2024

Título Escalabilidad y eficiencia en Inteligencia Artificial
a través de algoritmos federados verdes

Índice general

Peticionario Escuela Politécnica de Ingeniería de Ferrol
Rúa Mendizábal, s/n, Campus de Esteiro,
15403, Ferrol

Fecha Septiembre de 2024

Autor El Alumno

Fdo. Abel Pampín Rodríguez

Índice general

1 Índice general	1
Tablas	4
Figuras	5
2 Resumen	7
2.1 Resumen	8
2.2 Resumo	9
2.3 Summary	10
3 Memoria	11
3.1 Objeto	12
3.2 Alcance	12
3.3 Antecedentes	13
3.3.1 Inteligencia Artificial	13
3.3.2 Aprendizaje automático	14
Aprendizaje supervisado	15
Aprendizaje no supervisado	16
Aprendizaje por refuerzo	17
Desarrollo de modelos	18
3.3.3 Aprendizaje federado	27
Definición, características y proceso de desarrollo	28
Categorías y tipos de aprendizaje federado	30
Optimización	31
Amenazas y privacidad de datos	33
Casos de uso, <i>software</i> y <i>datasets</i>	36
3.3.4 Algoritmos verdes	36
IA verde y roja	37
Emisiones y huella de carbono de la IA	40
Iniciativas políticas europeas y nacionales	42
Notación Big-O	43
3.4 Diseño e implementación	46
3.4.1 Vista global del sistema	46
3.4.2 Sistemas informáticos	47
3.4.3 Sistemas operativos	51
3.4.4 Lenguajes de programación y librerías	51
3.5 Metodología	53
3.6 Análisis de soluciones	55
3.6.1 FedHEONN	55
Arquitectura y función objetivo	55

Aprendizaje federado y cifrado	58
Algoritmos coordinador y clientes	60
3.6.2 Agregación parcial e incremental	60
3.6.3 Ensamblaje	62
3.6.4 Paralelismo	68
3.6.5 Prototipo desarrollado para el entorno federado	70
3.7 Resultados finales	75
3.7.1 Agregación parcial e incremental	75
3.7.2 Ensamblaje	79
3.7.3 Paralelismo	81
3.7.4 Prototipo desarrollado para el entorno federado	84
3.7.5 Eficiencia: emisiones y consumo energético	88
3.8 Conclusiones y futuros trabajos	93
3.8.1 Conclusiones	93
3.8.2 Futuros trabajos	93
3.9 Referencias	95
3.9.1 Bibliografía	95
3.9.2 Otras referencias	98
4 Anexos	99
4.1 Documentación de partida	100
4.2 Código de programación	103
4.3 Configuración del entorno	105
4.4 Tablas y figuras de resultados	107

Índice de tablas

3.1	Órdenes más utilizados en análisis de algoritmos para la notación Big-O	43
3.2	Especificaciones de la Raspberry Pi Model 4B	49
3.3	Especificaciones de la Raspberry Pi Model Zero W	49
3.4	Especificaciones de la Jetson Nano	49
3.5	Especificaciones del Lenovo Legion Y540	51
3.6	Especificaciones del Lenovo ThinkPad X200	51
3.7	Sistemas operativos utilizados	51
3.8	Características de los conjuntos de datos	53
3.9	Resumen de los resultados experimentales de agregación parcial	75
3.10	Resultados del modelo con ensamblaje con los hiperparámetros óptimos obtenidos con validación cruzada	80
3.11	Tiempos de ejecución de los distintos procesos involucrados en el entrenamiento conjunto de FedHEONN	82
3.12	Hiperparámetros ejecutados en <code>codecarbon_mnist_incremental.py</code>	89
3.13	Desglose por tareas y modelo del consumo energético, emisiones, duración y exactitud de la primera prueba (sin encriptación)	89
3.14	Desglose por tareas y modelo del consumo energético, emisiones, duración y exactitud de la segunda prueba (con encriptación)	90
3.15	Hiperparámetros de los modelos en <code>codecarbon_mnist_experiments.py</code>	91
3.16	Consumo energético en experimento <code>codecarbon_mnist_experiments.py</code>	91
4.1	Algunos resultados del <i>grid-search</i> de <code>mnist_gridsearchcv_remote.py</code>	109
4.2	Métricas para regularizaciones usadas sin <i>ensemble</i> en MNIST	109
4.3	Algunos resultados del <i>grid-search</i> de <code>skin_gridsearchcv_remote.py</code>	110
4.4	Métricas para regularizaciones usadas sin <i>ensemble</i> en Skin	110
4.5	Algunos resultados del <i>grid-search</i> de <code>miniboone_gridsearchcv_remote.py</code>	111
4.6	Métricas para regularizaciones usadas sin <i>ensemble</i> en MiniBoone	111
4.7	Algunos resultados del <i>grid-search</i> de <code>drybean_gridsearchcv_remote.py</code>	112
4.8	Métricas para regularizaciones usadas sin <i>ensemble</i> en Dry Bean	112
4.9	Resultados <i>grid-search</i> de <code>carbon_nanotubes_gridsearchcv_remote.py</code>	113
4.10	Métricas para regularizaciones usadas sin <i>ensemble</i> en Carbon Nanotubes	113

Índice de figuras

3.1	Mapa conceptual IA - Ciencia de Datos	13
3.2	Flujo de procesado en métodos de Machine Learning	15
3.3	Resumen de las tres categorías principales en Aprendizaje automático [15]	15
3.4	Tareas de clasificación y regresión en aprendizaje supervisado [15]	16
3.5	Tareas de aprendizaje no supervisado: segmentación y reducción dimensional [15]	17
3.6	Flujo de trabajo de los métodos de aprendizaje por refuerzo [15]	18
3.7	Flujo de desarrollo típico de un modelo de <i>ML</i> [26]	19
3.8	Esquema de la división de datos en <i>sets</i> de entrenamiento, validación y test	21
3.9	División y uso del método de validación cruzada <i>K-fold</i> [22]	21
3.10	Gráficas simplificadas representando tres escenarios: <i>infraajuste</i> , <i>sobreajuste</i> y <i>ajuste deseado</i> [29]	22
3.11	Gráfica mostrando el proceso de sobreajuste basado en los errores de validación y entrenamiento [24]	22
3.12	Esquema de la matriz de confusión [4]	24
3.13	Ejemplo de resolución iterativa, por gradiente descendente, de una función de pérdida usada en regresión basada en el MSE [21]	26
3.14	Ajuste de hiperparámetros para un modelo de aprendizaje supervisado [18]	27
3.15	Ejemplo de aplicación, ciclo de vida y actores durante el entrenamiento de un sistema de Aprendizaje Federado [17]	28
3.16	Arquitecturas en sistemas de Aprendizaje Federado [34]	29
3.17	Estructura de datos compartidas en HFL [34]	31
3.18	Estructura de datos complementarios en VFL [34]	31
3.19	Comparativa de suma bajo esquema FHE y sin él [5]	35
3.20	Ley de Moore vs IA	38
3.21	Distintos modelos de detección de objetos en visión artificial: exactitud (acc) - FPO (billones) y número de parámetros del modelo (millones) [31]	39
3.22	Simulador web para la estimación del cálculo de <i>CO₂</i> equivalente de una RNA [19]	41
3.23	Informe generado por la herramienta CodeCarbon [7]	42
3.24	Gráfica del ratio de crecimiento y escalabilidad de las distintas clases de notaciones Big-O [1]	44
3.25	Ejemplo de arquitectura cliente-servidor para sistema federado	47
3.26	Raspberry Pi Model 4B	48
3.27	Raspberry Pi Zero W	48
3.28	Jetson Nano	50
3.29	Ordenadores portátiles disponibles	50
3.31	Endpoints para <i>ping</i> y <i>status</i> del servidor	71

3.32 Endpoints para la selección, carga y reparto de los datos de test y entrenamiento	71
3.33 Endpoints para la subida, selección, descarga y eliminación de contextos TenSEAL	72
3.34 Endpoints para el ajuste de hiperparámetros, sincronización de características y envío de pesos óptimos del coordinador	72
3.36 Endpoints para la <i>agregación parcial</i> y comprobación de la cola de procesado	73
3.37 Clase cliente para consumo de API, con inicialización, <i>ping</i> y <i>status</i>	74
3.38 Salida de <i>classification_incremental.py</i>	76
3.39 Salida de <i>classification_incremental_multiple.py</i>	76
3.40 Salida de <i>regression_incremental.py</i>	77
3.41 Salida de <i>regression_incremental_multiple.py</i>	77
3.42 Salida de <i>mnist.py</i>	78
3.43 Tabla con el rango paramétrico para los distintos <i>grid-search</i> ejecutados	79
3.44 Tiempos para las distintas tareas involucradas en FedHEONN	83
3.45 Arranque del servidor.	84
3.46 Configuración utilizada durante la <i>demo</i> para cliente y coordinador FedHEONN.	85
3.47 Ejecución de <i>demo_server.py</i> y reflejo en el <i>status</i> del servidor.	86
3.48 Descarga de los datos del conjunto de entrenamiento desde el servidor.	86
3.49 Arranque de los clientes.	87
3.50 Comienzo del entrenamiento local en ambos clientes.	87
3.51 Funcionamiento de la cola de agregación en el servidor.	87
3.52 Finalización de ambos clientes con predicción sobre <i>test</i>	88
4.1 ① - Configuración e instalación de SO en dispositivos Raspberry	105
4.2 ② - Configuración e instalación de SO para Jetson Nano	105
4.3 ③ - Conexión mediante SSH y actualización e instalación de programas y librerías	106
4.4 Salida del programa <i>mnist_gridsearchcv_remote.py</i>	107
4.5 Salida del programa <i>skin_gridsearchcv_remote.py</i>	107
4.6 Salida del programa <i>drybean_gridsearchcv_remote.py</i>	108
4.7 Salida del programa <i>carbon_gridsearchcv_remote.py</i>	108
4.8 Salida de <i>mnist_parallel.py</i> en serie.	114
4.9 Salida de <i>mnist_parallel.py</i> en paralelo.	115

Título Escalabilidad y eficiencia en Inteligencia Artificial
a través de algoritmos federados verdes

Resumen

Peticionario Escuela Politécnica de Ingeniería de Ferrol
Rúa Mendizábal, s/n, Campus de Esteiro,
15403, Ferrol

Fecha Septiembre de 2024

Autor El Alumno

Fdo. Abel Pampín Rodríguez

A continuación se incluye una breve descripción, a modo de resumen, en castellano, gallego e inglés, del proyecto realizado para este Trabajo Final de Grado.

2.1. Resumen

En los últimos años los modelos de Inteligencia Artificial (IA) han aumentado de tamaño en pasos agigantados, aupados por unos resultados y casos de uso cada vez mejores y más refinados; con todo, los algoritmos actuales de estos modelos se centran, como único objetivo, en conseguir la máxima eficacia en la resolución del problema asignado, pero no tienen en cuenta la eficiencia energética del modelo conjunto. Se ha comprobado que estos modelos, y la infraestructura necesaria para ellos, tienen un impacto muy importante en el consumo de recursos eléctricos e hídricos, por lo que es necesario incluir este enfoque en su desarrollo y reducir tales impactos.

Un planteamiento reciente en la IA es el llamado Aprendizaje Federado, que permite entrenar modelos de manera colaborativa en múltiples dispositivos sin necesidad de centralizar sus datos y preservando su privacidad.

Dicha técnica se presenta como una alternativa que solventa las desventajas de las técnicas centralizadas corrientes, que requieren grandes silos de información y un mayor consumo energético y temporal para su entrenamiento. El Aprendizaje Federado, por otro lado, es distribuido, eficiente y privativo por diseño, por lo que es un incentivo para la convergencia de las nuevas tecnologías de la información como *IoT* y las operativas propias de la *Industria 4.0*.

En este Trabajo Final de Grado se ha abordado el desarrollo, análisis y experimentación de métodos de aprendizaje federado desde una perspectiva verde, en sintonía con la manifestación de interés del Programa Nacional de Algoritmos Verdes [8] publicado por el Ministerio de Asuntos Económicos y Transformación Digital [9]. Específicamente, hemos contribuido y mejorado el método de entrenamiento regularizado y encriptado para redes neuronales de una sola capa que ha sido desarrollado por la UDC en entornos federados (*FedHEONN*) [12], y experimentado con su implementación en dispositivos embebidos y simulados.

Como tal, hemos aplicado un método de ensamblaje conocido como Random Patches al algoritmo para contrastar su rendimiento frente la versión original, además de implementar la capacidad de realizar un entrenamiento incremental por lotes, con el objetivo de lograr un algoritmo más versátil y de mayor capacidad de representación; obteniendo resultados satisfactorios y acompañando estos apartados con un estudio de la eficacia de estas técnicas en comparación con su versión original y otro tipo de modelos.

Por último, desarrollamos también la paralelización de ciertos procesos clave del algoritmo de entrenamiento de FedHEONN, con el objetivo de reducir los tiempos globales de ejecución, a la par que se construye una API para servidor y cliente/s que proporciona los medios para su uso entre varios dispositivos conectados a la red, simulando un escenario federado realista.

2.2. Resumo

Nos últimos anos os modelos de Intelixencia Artificial (IA) aumentaron de tamaño en pasos axigantados, levados por uns resultados e casos de uso cada vez mellores e máis refinados; con todo, os algoritmos actuais destes modelos céntranse, como único obxectivo, en conseguir a máxima eficacia na resolución do problema asignado, pero non teñen en conta a eficiencia enerxética do modelo conxunto. Comprobouse que estes modelos, e a infraestrutura necesaria para eles, teñen un impacto moi importante no consumo de recursos eléctricos e hídricos, polo que é necesario incluír este enfoque no seu desenvolvemento e reducir tales impactos.

Unha formulación recente na IA é a chamada Aprendizaxe Federada, que permite adestrar modelos de maneira colaborativa en múltiples dispositivos sen necesidade de centralizar os seus datos e preservando a súa privacidade.

Dita técnica preséntase como unha alternativa que resolve as desvantaxes das técnicas centralizadas correntes, que requiren grandes silos de información e un maior consumo enerxético e temporal para o seu adestramento. A Aprendizaxe Federada, doutra banda, é distribuído, eficiente e privativo por deseño, polo que é un incentivo para a converxencia das novas tecnoloxías da información como *IoT* e as operativas propias da *Industria 4.0*.

Neste Traballo Final de Grao abordouse o desenvolvemento, análise e experimentación con métodos de aprendizaxe federada dende unha perspectiva verde, en sintonía coa manifestación de interese do Programa Nacional de Algoritmos Verdes [8] publicado polo Ministerio de Asuntos Económicos e Transformación Dixital [9]. Especificamente, contribuímos e melloramos o método de adestramento regularizado e encriptado para redes neuronais dunha soa capa desenvolto pola UDC en contornas federadas (*FedHEONN*) [12], e experimentamos coa súa implementación en dispositivos embebidos e simulados.

Como tal, aplicamos un método de ensamblaxe coñecido como Random Patches ao algoritmo para contrastar o seu rendemento fronte a versión orixinal, ademais de implementar a capacidade de realizar un adestramento incremental por lotes, co obxectivo de lograr un algoritmo más versátil e de maior capacidade de representación; obtendo resultados satisfactorios e acompañando estes apartados cun estudo da eficacia destas técnicas en comparación coa versión orixinal e outro tipo de modelos.

Por último, desenvolvemos tamén a paralelización de certos procesos clave do algoritmo de adestramento de *FedHEONN*, co obxectivo de reducir os tempos globais de execución, á vez que se constrúe unha API para servidor e cliente/s que proporciona os medios para o seu uso entre varios dispositivos conectados á rede, simulando un escenario federado realista.

2.3. Summary

In recent years, Artificial Intelligence (AI) models have increased in size immensely, boosted by increasingly better and more refined results and use cases; however, the current algorithms of these models focus, as their sole objective, on achieving maximum effectiveness in solving the assigned problem, but do not take into account the energy efficiency of the joint model. It has been found that these models, and the infrastructure required for them, have a very significant impact on the consumption of electrical and water resources, so it is necessary to include this approach in their development in order to reduce such impacts.

A recent approach in AI is the so-called Federated Learning, which allows models to be trained collaboratively on multiple devices without the need to centralize their data and preserving their privacy.

This technique is presented as an alternative that overcomes the disadvantages of current centralized techniques, which require large information silos and greater energy and time consumption for its training. Federated Learning, on the other hand, is distributed, efficient and privative by design, making it an incentive for the convergence of new information technologies such as IoT and Industry 4.0 operations.

This Final Degree Project has addressed the development, analysis and experimentation of federated learning methods within a green perspective, in line with the manifestation of interest of the National Green Algorithms Program [8] published by the Ministry of Economic Affairs and Digital Transformation [9]. Specifically, we have contributed to and improved the regularized and encrypted training method for single-layer neural networks that has been developed by UDC in federated environments (*FedHEONN*) [12], and experimented with its implementation on embedded and simulated devices.

As such, we have applied an ensemble method known as Random Patches to the algorithm to contrast its performance against the original version, in addition to implementing the ability to perform incremental batch training, with the aim of achieving a more versatile algorithm and greater representation capacity, obtaining satisfactory results and accompanying these sections with a study of the efficiency of these techniques compared to its original version and other types of models.

Finally, we also developed the parallelization of certain key processes of the FedHEONN training algorithm, with the aim of reducing the overall execution times, while building an API for server and client/s that provides a medium for its use among several devices connected to the network, simulating a realistic federated scenario.

Título Escalabilidad y eficiencia en Inteligencia Artificial
a través de algoritmos federados verdes

Memoria

Peticionario Escuela Politécnica de Ingeniería de Ferrol
Rúa Mendizábal, s/n, Campus de Esteiro,
15403, Ferrol

Fecha Septiembre de 2024

Autor El Alumno

Fdo. Abel Pampín Rodríguez

3.1. Objeto

En este Trabajo Final de Grado (TFG) se realizará la investigación y desarrollo de un método de aprendizaje federado que pueda ser implementado en dispositivos embebidos, con un triple propósito:

- Desarrollar un método de ensamblaje basado en un algoritmo federado, comparando los resultados obtenidos con la versión original para distintos conjuntos de datos utilizados comúnmente como referentes en este tipo de literatura.
- Implementar un prototipo en una arquitectura cliente servidor que permita recrear un escenario realista federado entre distintos dispositivos físicos y/o simulados conectados a la red.
- Realizar un análisis del consumo energético y del impacto medioambiental de las distintas técnicas empleadas para su posterior estudio y equiparación con equivalentes centralizadas.

Este último punto recoge la intención de fomentar el desarrollo de una Inteligencia Artificial verde por diseño (Green by Design) y estimular este factor como un criterio más a tener en cuenta a la hora de decidir el uso de uno u otro método en el campo del Aprendizaje Automático. El uso de las tecnologías de la información y comunicaciones supone hoy en día entre un 5 % y 9 % del consumo total de electricidad en el mundo, y según un informe realizado por el Parlamento Europeo [27], podría llegar al 20 % en 2030. Dentro de dicho desglose, se cita que los recursos de computación necesarios para entrenar modelos de inteligencia artificial se están doblando cada cuatrimestre desde el año 2012, a medida que se buscan modelos cada vez más precisos y eficaces. Esto obliga a concienciar y acompañar tales avances tecnológicos con una perspectiva medioambientalmente sostenible.

3.2. Alcance

El ámbito de aplicación de este Trabajo Final de Grado aborda diversos aspectos dentro del llamado aprendizaje automático y requiere el uso de distintos dispositivos embebidos. Para ello es necesario:

- **Comprendión** del estado del arte de la inteligencia artificial, más concretamente del aprendizaje automático y federado.
- **Estudio** de la técnica y algoritmos de partida para evolucionar posibles mejoras y aplicaciones.
- **Desarrollo** en Python de un algoritmo de aprendizaje federado con técnicas de ensamblaje, y su posterior despliegue en la red federada conformada tanto por dispositivos físicos como simulados.
- **Familiarización**, configuración e implantación de los sistemas desarrollados en dispositivos embebidos.
- **Experimentación** para contrastar la eficacia y eficiencia de las nuevas mejoras.
- **Análisis** de los resultados experimentales y conclusiones finales.
- **Documentación** del TFG.

3.3. Antecedentes

Planteamos esta sección con varios objetivos: primero realizaremos una introducción a la inteligencia artificial y al desarrollo de modelos, para luego adentrarnos en una tendencia reciente conocida como Aprendizaje Federado (Federated Learning, FL), y su relación con lo que se denominan *algoritmos verdes*.

3.3.1. Inteligencia Artificial

Antes de entrar de lleno en las diferentes técnicas y algoritmos que ofrece la inteligencia artificial, es bueno dar un repaso por los fundamentos de este campo y sus diferentes ramas. La mejor forma de introducir este tema es tener bien claro a qué nos estamos refiriendo; ¿qué es aquello a lo que se llama *Artificial Intelligence*?; ¿cómo se distingue de otros conceptos como *Machine Learning*, *Data Mining*, *Big Data* y otros conceptos tan en boga hoy día?

A continuación presentamos un mapa conceptual, figura 3.1, y unas definiciones que ayudan a aclarar estas nociones:

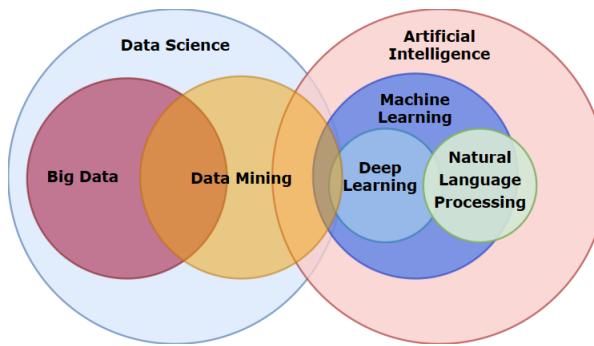


Figura 3.1: Mapa conceptual IA - Ciencia de Datos

Inteligencia Artificial (*Artificial Intelligence*, AI) disciplina que busca crear algoritmos y sistemas que puedan imitar a la inteligencia humana para realizar tareas complejas de forma autónoma, por ejemplo percepción, toma de decisiones o aprendizaje.

Aprendizaje Automático (*Machine Learning*, ML) conjunto de algoritmos dentro del ámbito de la inteligencia artificial que son capaces de identificar y reconocer patrones en conjuntos de datos para realizar predicciones u otras tareas, descubriendo, por si solos, relaciones entre las variables sin haber sido programados explícitamente para ello, si no a base de tratar con un gran volumen de datos o ejemplos.

Machine Learning es un término originado en los años 1950, definido por primera vez por Arthur Lee Samuel, informático teórico estadounidense. Tom Mitchell, informático teórico posterior, llegó a proponer la siguiente definición, más precisa: *Un programa se dice que aprende de la experiencia E con respecto a una tarea T y una métrica de desempeño P, si su rendimiento en T, medido por P, mejora con la experiencia E.*

De estas dos definiciones podemos concluir que el Aprendizaje Automático es una manera de Inteligencia Artificial. De todos modos, podemos tener AI sin ML; por ejemplo, si

codificamos reglas, árboles de decisión o técnicas de búsqueda, podemos crear un agente heurístico de inteligencia artificial, sin un enfoque basado en el del ML.

Aprendizaje Profundo (Deep Learning, DL) técnica de ML que imita redes neuronales de manera artificial para aprender a partir de datos no estructurados, consiguiendo una mayor capacidad de representación que otras técnicas de ML tradicional.

Ciencia de Datos (Data Science) campo interdisciplinario que utiliza métodos estadísticos, de programación y de análisis de datos para extraer conocimiento de conjuntos de datos.

Grandes conjuntos de datos (Big Data) término usado para el almacenamiento y procesando de grandes volúmenes de datos.

Minería de Datos (Data Mining), subconjunto de la Ciencia de Datos especializado en los procesos de extracción de información útil de grandes conjuntos de datos en bruto estructurados y heterogéneos.

Las técnicas exploradas y desarrolladas en este Trabajo Final de Grado corresponden a un área dentro del *Aprendizaje Automático* relativamente reciente, llamado *Aprendizaje Federado*, por lo que las siguientes secciones se centrarán en profundizar en los fundamentos de ambas, así como en las técnicas de procesado de datos y preparación de los mismos, que son tan fundamentales para un buen desempeño como la idoneidad o eficacia del propio algoritmo.

3.3.2. Aprendizaje automático

El aprendizaje automático nace con la idea de buscar algoritmos que hagan posible que los ordenadores puedan *aprender* – sin ser programados para seguir unas reglas específicas dictadas de antemano – a través de la experiencia; partiendo de un gran volumen de ejemplos.

Estos ejemplos formarán lo que se conoce luego como datos de entrada y, más específicamente, predictores o características (*features or predictors*). Así, los algoritmos de ML se usan para descubrir patrones ocultos, establecer relaciones y grupos o predecir algún atributo.

De lo anterior se puede concluir que el *Machine Learning* se puede usar para la resolución de problemas que no estén determinados con fórmulas precisas y que, por tanto, se podrían reducir a un planteamiento basado en sistemas de reglas; si no para aquellos donde se desconozca la correspondencia entre causas y efectos, bien por la gran cantidad de variables dependientes, bien por la complejidad o aleatoriedad de las mismas.

Las técnicas de ML hacen un uso conjunto de distintas áreas de conocimiento para lograr sus objetivos, como la estadística, probabilidad, cálculo y optimización mediante métodos numéricos. Además, también se inspira en otras disciplinas como la teoría de la información, la psicología o la neurociencia.

Antes de introducir el tipo de aprendizaje automático que nos incumbe, el caso **supervisado**, que abordaremos más adelante, se va a introducir ahora la terminología necesaria y también una aproximación general al funcionamiento de un método típico de ML.

La figura 3.2 ilustra, de forma simplificada, la terminología y la metodología empleada en el flujo de trabajo de un método genérico de *aprendizaje automático*, donde se recogen unos datos brutos, se procesan, y son alimentados al modelo divididos en conjuntos para su entrenamiento, validación y optimización. Una vez satisfechos con el rendimiento y eficacia dadas por las métricas de evaluación se considera el modelo listo y se procede a su despliegue.

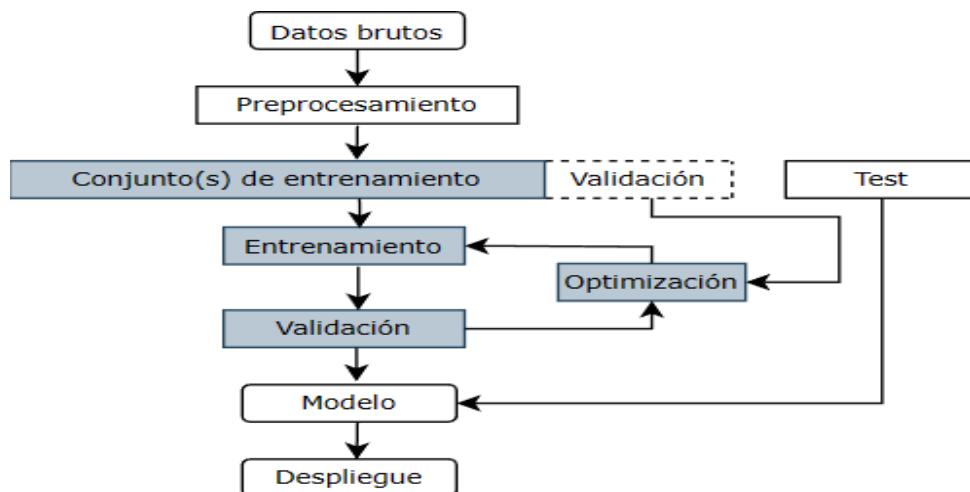


Figura 3.2: Flujo de procesado en métodos de Machine Learning

Existen varios tipos de aprendizaje automático, pero sin duda hay tres grandes categorías, el **aprendizaje supervisado**, el **no supervisado** y **por refuerzo**. La figura 3.3 muestra esta división y las entradas y salidas típicas correspondientes para cada una de ellas.

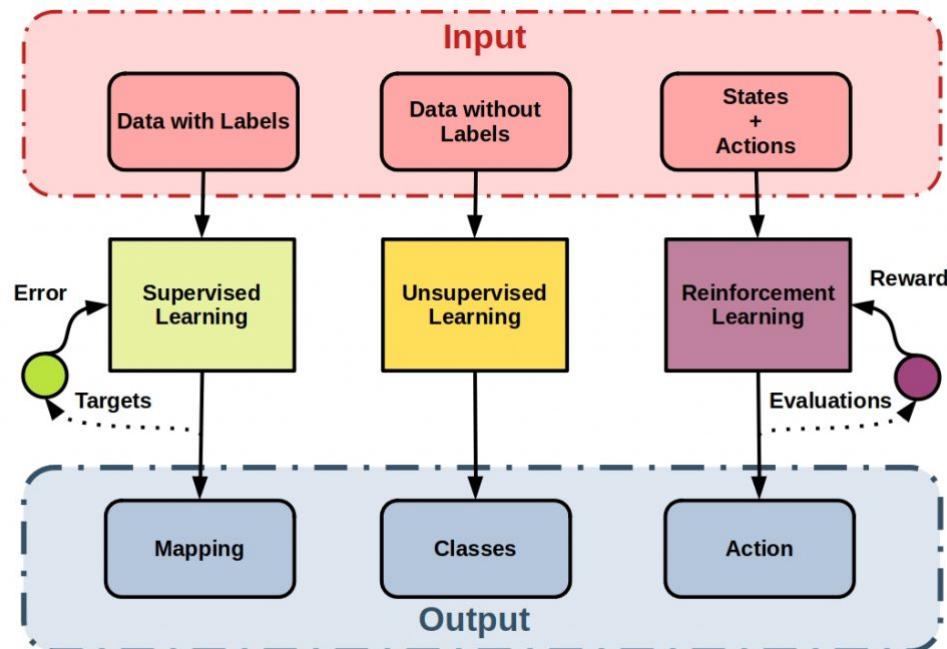


Figura 3.3: Resumen de las tres categorías principales en Aprendizaje automático [15]

Aprendizaje supervisado

Como sugiere el nombre, el aprendizaje para estos métodos está supervisado porque se basa en un atributo u objetivo específico del que se tiene constancia, es decir, que se encuentra correctamente etiquetado.

Así, el conjunto de datos consta de un atributo objetivo etiquetado que se encuentra acompañado de otras variables que se usarán para predecirlo, denominadas *predictores* o *características*.

El objetivo puede ser un atributo numérico continuo o un atributo de clases para representar múltiples categorías. Basado en la anterior división, se habla entonces de tareas de aprendizaje supervisado de **regresión** o **clasificación**, respectivamente.

Como se muestra en la parte superior de la figura 3.4, la regresión intenta ajustar los datos a una serie de valores continuos, mientras que la clasificación, en la parte inferior, trata de separar el conjunto de datos en clases.

Existen muchos algoritmos de aprendizaje automático dentro de esta categoría, por mencionar algunos de ellos:

- Regresión logística.
- Regresión lineal.
- Árboles de decisión.
- K-vecinos más cercanos.
- Bosques aleatorios.
- Máquinas de Vectores soporte.
- Redes de neuronas artificiales (RNA).

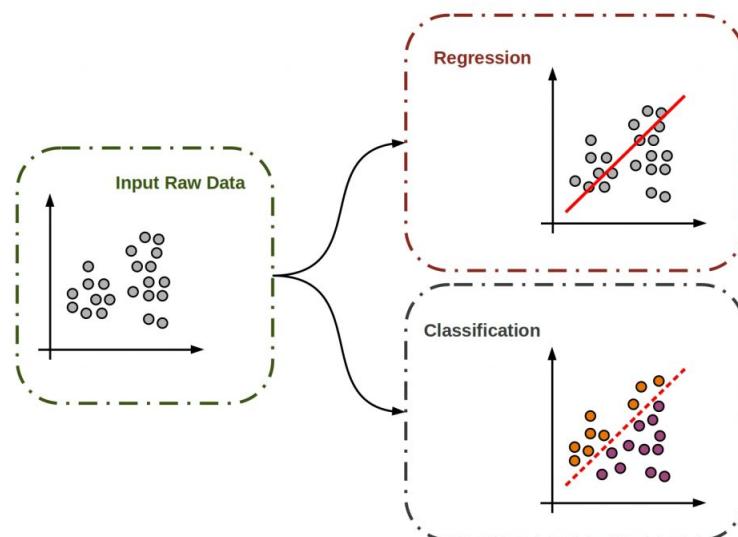


Figura 3.4: Tareas de clasificación y regresión en aprendizaje supervisado [15]

Aprendizaje no supervisado

En el caso del aprendizaje no supervisado, no existe un atributo objetivo, es decir, no se cuenta con datos etiquetados. El fin de estos métodos es el de identificar patrones deduciéndo la estructura y relaciones que tienen los atributos del conjunto de datos entre sí, separándolos en diferentes grupos o *clusters*. Existen dos grandes tipos de problemas que resuelven estas técnicas, la segmentación (*clustering*) y la reducción dimensional.

En la mitad superior de la figura 3.5 se ilustra cómo la segmentación agrupa los datos en diversos conjuntos según características similares que encuentre entre ellos, mientras que, en

la mitad inferior, se aprecia cómo la reducción dimensional intenta comprimir, en un número menor de variables, e intentando perder la mínima información posible, el conjunto de datos.

Entre las aplicaciones prácticas típicas de estas técnicas estarían la agrupación de clientes, recomendación de productos y extracción de componentes principales.

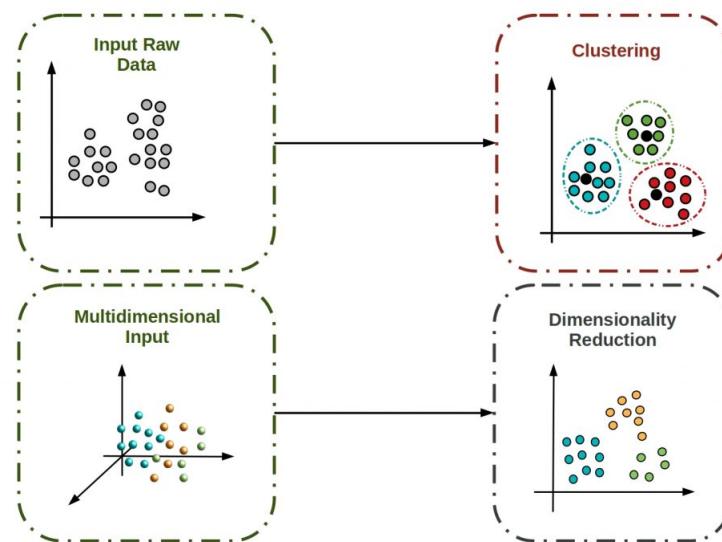


Figura 3.5: Tareas de aprendizaje no supervisado: segmentación y reducción dimensional [15]

Algunos de los algoritmos usados para este fin son:

- K-medias.
- Clustering espectral.
- Clustering jerárquico.
- Análisis de componentes principales (*Principal Component Analysis, PCA*).
- Descomposición en valores singulares (*Singular Value Decomposition, SVD*).

Aprendizaje por refuerzo

En esta categoría de aprendizaje automático existe un agente que interactúa con un entorno y que aprende la toma de decisiones que maximice una recompensa numérica dada por el modelo. Como se observa en la figura 3.6, el agente *aprende* a través de la prueba y error de manera iterativa, recibiendo información sobre el éxito o fracaso de sus *acciones* mediante *recompensas* o penalizaciones, de forma que existe una retroalimentación entre agente y el entorno.

Los conceptos clave en el aprendizaje por refuerzo (Reinforcement Learning, RL) son el agente (algoritmo ML), el entorno (espacio de problemas adaptativo), la acción (paso que el agente realiza para navegar por el entorno), el estado (situación del entorno en un momento dado), la recompensa o castigo (recibido por realizar una acción) y la recompensa acumulada (suma de todas ellas). El objetivo es así el de encontrar una estrategia que maximice la recompensa acumulada a largo plazo ya que, en general, existe siempre una mediación entre la exploración del entorno, la selección de acciones y la actualización de la política o función de valor correspondiente.

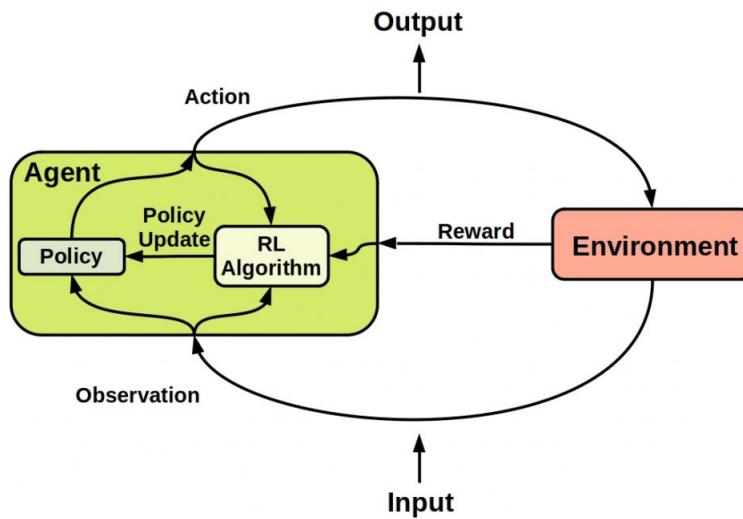


Figura 3.6: Flujo de trabajo de los métodos de aprendizaje por refuerzo [15]

En cuanto a los algoritmos típicos de esta categoría de aprendizaje, tenemos:

- Aprendizaje-Q.
- Método de gradiente de políticas.
- Métodos Montecarlo.
- Aprendizaje por diferencia temporal.

Todos ellos se pueden agrupar en dos categorías, el RL *basado en modelos*, cuando el entorno está bien definido e invariable, y el RL *sin modelo*, cuando el entorno es grande, complejo, desconocido o cambiante. Este tipo de métodos se usan comúnmente en problemas de toma de decisiones secuenciales y condicionales.

Desarrollo de modelos

En esta sección abordamos conceptos generales de *Aprendizaje Automático* que se han de tener en cuenta a la hora de desarrollar un modelo, además de los pasos y flujos necesarios para llevarlos a cabo.

En general, el desarrollo de un modelo de **ML** implica siempre las siguientes fases, tal y como se ilustran en la figura 3.7:

1. **Definir** y entender el problema.
2. **Recopilar** datos.
3. **Preparar** datos.
4. **Seleccionar** el algoritmo apropiado y construir el modelo.
5. **Evaluar** y optimizar el modelo.
6. **Desplegar** para uso práctico.



Figura 3.7: Flujo de desarrollo típico de un modelo de *ML* [26]

① Definición del problema

Es muy importante empezar entendiendo el problema a resolver y las necesidades y características del mismo; *¿que objetivo buscamos?*. Para lograr una aplicación y uso satisfactorio hay que definir bien los datos disponibles, el tipo de problema y los requisitos de su solución.

Así, los datos disponibles pueden estar en varias localizaciones, generarse en tiempo real, presentar múltiples formatos y necesitar más o menos preprocesamiento. El tipo de problema puede ser uno de predicción de valores, clasificación, agrupación, reconocimiento de voz, imágenes, etc. Los requisitos de la solución son aquellos requerimientos de negocio que necesite cumplir el modelo final, ya que no sirve de nada construir un modelo que luego no pueda cumplir en cuestión de tiempo de respuesta/disponibilidad/veracidad las necesidades iniciales.

② Recopilación de datos

La recogida de datos de una o varias fuentes necesita satisfacer los requerimientos del problema y del modelo, que puede llegar a necesitar un gran volumen de datos representativos para ser capaz de descubrir los patrones y las relaciones entre ellos.

El formato de los datos y la estructura precisada para posteriores fases es trabajo para este ciclo del desarrollo, que muchas veces echa mano de técnicas de grandes conjuntos de datos (*Big Data*) para lograrlo.

③ Preparación de datos

La preparación de datos o **preprocesamiento** de datos es uno de los pasos críticos, si no el que más, a la hora de construir un modelo de *Aprendizaje Automático* eficaz, ya que no sólo dispone el conjunto de datos de tal modo que puedan ser alimentados al modelo, si no que también se encarga de su tratamiento para que estos arrojen unos resultados satisfactorios, adaptándose a las necesidades del modelo y a los requisitos de negocio.

El preprocesamiento de datos alberga dos grandes fases, que son la **transformación de datos** y la **selección/generación de características**. A grandes rasgos, la primera ataña, para *valores numéricos*, su escalado, tratamiento de datos faltantes y atípicos y su agrupamiento,

mientras que para *valores categóricos* concierne su codificación y tratamiento de valores faltantes. La segunda fase trata de seleccionar aquellos atributos y características representativas del problema, filtrar los datos incompletos o generar datos nuevos si fuese necesario.

Esta preparación de datos incumbe toda una serie de pasos que se describen – superficialmente – a continuación:

1. **Limpieza de datos:** consiste en eliminar datos incompletos, irrelevantes y, en general, la transformación del conjunto a un formato uniforme y coherente.
2. **Exploración de datos:** se realiza un primer análisis del conjunto de datos para visualizar patrones, tendencias, correlaciones entre variables e identificar valores atípicos o *outliers* (valores que se salen de una medida del rango intercuartílico adoptada), los cuales se suelen o bien eliminar del conjunto o bien sustituir por datos acotados (técnicas *trimming/winsorizing* respectivamente).
3. **Selección de características:** efectuado el anterior análisis, se deberían de filtrar ahora las características/atributos más relevantes y significativos, y descartar aquellos cuya correlación es nula o irrelevante, para ayudar a la convergencia en el entrenamiento del modelo y mejorar así su precisión en las predicciones.
4. **Tratamiento de valores faltantes (*missing-values*):** en gran parte de escenarios se encuentran observaciones donde faltan alguna de las características muestreadas, por lo que se suele llevar a cabo una de las siguientes acciones: eliminación de la observación o reemplazo por una constante global/media/mediana.
5. **Balanceamiento de datos:** repercute, sobre todo, a problemas de clasificación en aprendizaje supervisado, ya que es común encontrarse con datos en los que una de las clases tiene menos muestras que el resto. Estos casos de conjuntos desequilibrados se suele remediar mediante métodos como el *down-sampling* o *over-sampling*. El primero reduce el número de muestras de las clases mayoritarias y el segundo aumenta el número de muestras de la clase minoritaria (duplicando o creando datos nuevos) hasta equilibrarlas.
6. **Codificación de datos:** en este paso se realiza la transformación de datos para su entrada al modelo y mejorar así su rendimiento. Incluye etapas de escalado y normalización, usando técnicas como *estandarización Z-score* o *normalización min-max*, además de realizar la codificación de datos categóricos, usando por ejemplo *one-hot encoding*, o, también, el agrupamiento o *binning* de valores numéricos, incluyendo además posibles transformaciones logarítmicas o exponenciales que ayuden al modelo.
7. **División de datos:** paso fundamental y necesario en el desarrollo de todo modelo, y por el cual se fragmenta aleatoriamente el conjunto de datos preprocesado en tres conjuntos, **entrenamiento, validación y test**, en unas determinadas proporciones, normalmente un 60 % – 20 % – 20 % correspondientemente. El modelo aprende con los datos de entrenamiento, optimiza configuraciones y evalúa con los de evaluación y prueba con los de test.

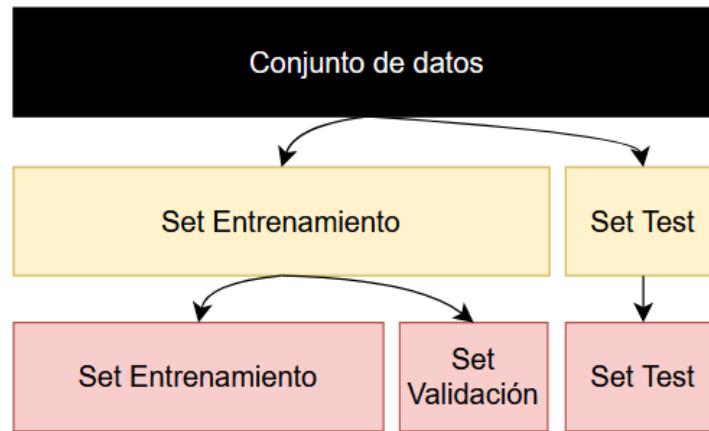


Figura 3.8: Esquema de la división de datos en *sets* de entrenamiento, validación y test

La división en conjuntos de entrenamiento y test independientes sería suficiente para evaluar la precisión del modelo, pero no se asegura así que se haya generalizado bien el problema y no simplemente 'memorizado' las salidas necesarias.

Un método popular y más robusto todavía de división de datos es el llamado de **validación cruzada**, y una de sus técnicas más populares se conoce por *K-fold cross-validation*, donde el set de entrenamiento se divide en k submuestras de igual tamaño. Durante el entrenamiento, $k - 1$ muestras se usan para entrenar el modelo y la restante para validar. Esto se repite k veces con cada una de las submuestras restantes, de manera que se garantiza que los datos de entrenamiento y validación se van rotando durante la optimización del modelo. Los resultados de la evaluación se combinan mediante media aritmética para conseguir la estimación oportuna, como se ilustra en la figura 3.9.

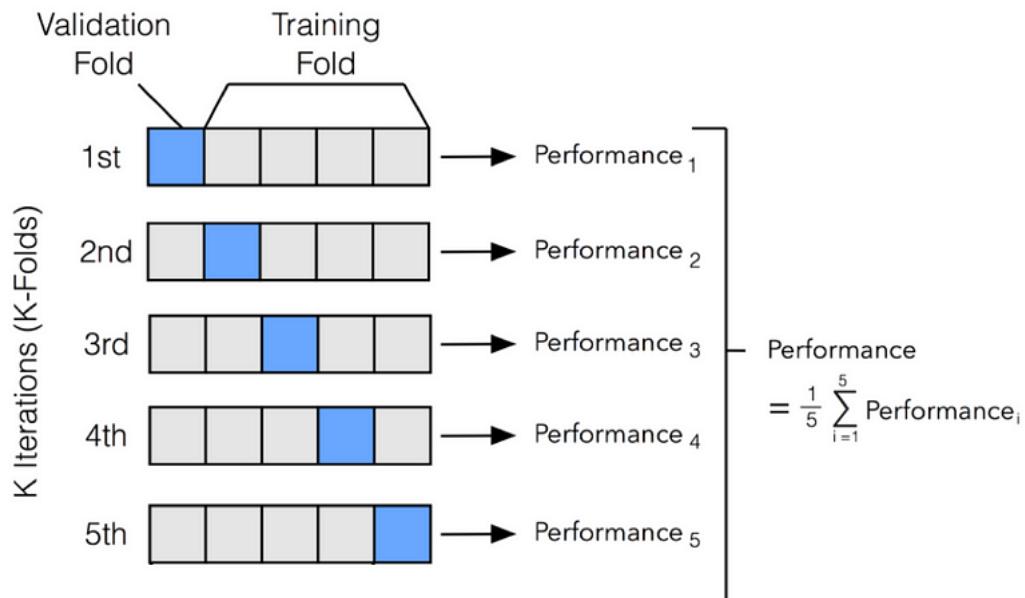


Figura 3.9: División y uso del método de validación cruzada *K-fold* [22]

Con esta división o *preprocesamiento* del conjunto de datos se pretende que el *dataset* obtenido sea **representativo**, para obtener un modelo eficaz y sacar predicciones útiles del mismo. Esto significa, entre otras cosas, evitar el **overfitting** o sobreajuste y el **underfitting** o infraajuste del modelo, y escapar así del conflicto que existe entre *optimización* y *generalización*. El sobreajuste ocurre cuando se entrena demasiado un modelo del cual se conocen sus salidas deseadas, 'memorizando' pautas que son demasiado específicas de los datos de entrenamiento e irrelevantes para datos nuevos, como se aprecia en la figura 3.10. Infraajuste es cuando ocurre todo lo contrario, y el modelo carece de capacidad de representación para ajustarse a las necesidades del *dataset*.

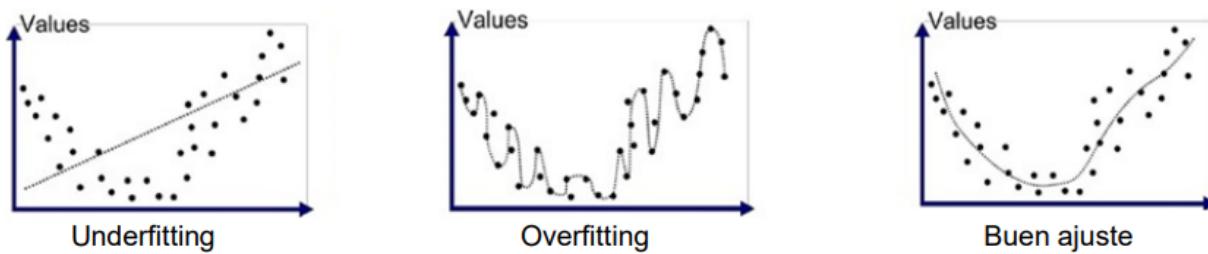


Figura 3.10: Gráficas simplificadas representando tres escenarios: *infraajuste*, *sobreajuste* y *ajuste deseado* [29]

Un indicador de que se está produciendo sobreajuste es cuando, luego de descender ambos, el error del conjunto de validación empieza a crecer, o estancarse, mientras que el error sobre el conjunto de entrenamiento sigue bajando, como se observa en la figura 3.11.

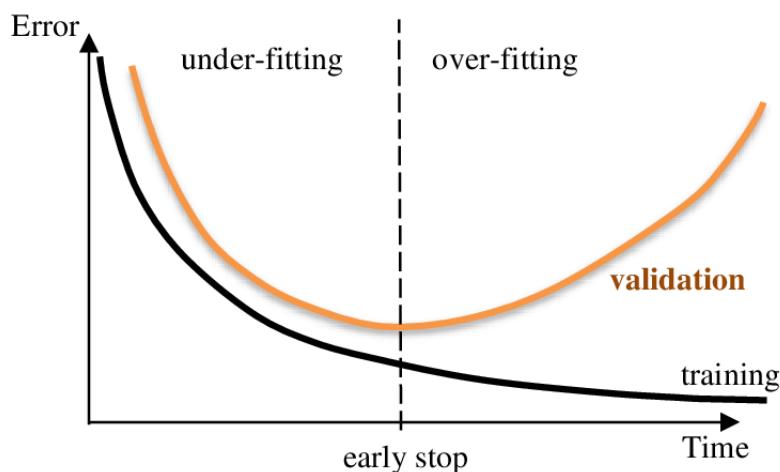


Figura 3.11: Gráfica mostrando el proceso de sobreajuste basado en los errores de validación y entrenamiento [24]

Para evitar el sobreajuste existen técnicas sencillas y complejas. La más sencilla de todas es la llamada *early-stopping*, que significa detener el entrenamiento una vez que se observan tendencias distintas en los errores de la figura 3.11. Otra de fácil implementación sería la propia *validación cruzada* que se explicó antes. Por último, la llamada **regularización** consiste en reducir la complejidad global del modelo, a costa de su varianza. Técnicas de este estilo incluyen la regularización *L1*, *L2*, *Lasso* o *Ridge*.

④ Selección y construcción del modelo

A estas alturas, la selección del algoritmo apropiado y el tipo, estructura y conjunto de datos de trabajo que este necesita ya debería de estar listo, puesto que su tratamiento, división e incluso transformación – muchas veces específica al algoritmo, ya que puede funcionar mejor con según que tipo de entradas – ya se ha abordado en las fases de preparación de datos.

Así, este aspecto de construcción del modelo implica el diseño de su arquitectura y puesta en marcha.

En cuanto a su diseño, se puede optar por arquitecturas centralizadas o distribuidas. Un diseño centralizado implica un modelo donde todo el procesamiento de datos – y su entrenamiento – se realiza en un solo lugar, como una única máquina o servidor. Por la contra, las arquitecturas distribuidas dividen este proceso en varias máquinas o servidores, lo que permite realizar su tratamiento en paralelo y en diferentes lugares geográficos. Este último enfoque se ha de tener en cuenta si existe una necesidad de procesar grandes cantidades de datos en un tiempo razonable, además de si, por la naturaleza del problema, la recogida de datos y su almacenamiento presenta límites físicos o incluso legales que se han de considerar. El diseño también puede llevar consigo el propio dimensionamiento del modelo, sobre todo cuando se trabaja con técnicas de Aprendizaje Profundo (*Deep Learning*, DL), donde la decisión del número de capas ocultas, el número de neuronas por capa y las funciones de activación asignadas a estas es una parte tan fundamental como el propio conjunto de datos y la solución deseada.

Por último, la puesta en marcha de un modelo no garantiza su eficacia ni éxito, y muchas veces es necesario probar con distintos algoritmos y diseños para quedarse con el que presente mejores resultados y desempeño.

⑤ Evaluación y optimización

La evaluación y optimización del modelo significa la introducción de las métricas de desempeño, la función de coste o pérdida y la afinación de hiperparámetros pero, ¿que son? y, ¿para que sirven?:

Métricas Las métricas de desempeño son una serie de indicadores del rendimiento del modelo, de su eficacia, de como de 'bueno' es y, en general, de qué cantidad de error ha cometido cuando se evalúa contra el conjunto de datos de test o de validación.

Existen muchas métricas de desempeño diferentes, pero generalmente se suelen dividir según el tipo de método de aprendizaje automático que se utilice. Así, por ejemplo, hay métricas propias de técnicas de aprendizaje supervisado, no-supervisado y por refuerzo. A continuación desglosamos las más utilizadas para el caso supervisado, que es el que nos incumbe en este trabajo.

Aprendizaje supervisado

Dado un conjunto de datos D de n muestras, $y \in \mathbb{R}^n$ el vector cuyas componentes y_j representan las observaciones o salidas reales de dicho conjunto, \bar{y} el promedio de los valores anteriores e $\hat{y} \in \mathbb{R}^n$ el vector cuyas componentes \hat{y}_j son las predicciones resultante del modelo. Para evaluar modelos de **regresión** se suelen emplear las siguientes métricas:

- MAE, *Mean Absolute Error*, es el error absoluto medio entre los valores reales y la predicción.

$$MAE = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j|$$

- RMSE, *Root Mean Squared Error*, es el error cuadrático medio entre observaciones y predicciones.

$$RMSE = \sqrt{\frac{1}{n} \sum_{j=1}^n (y_j - \hat{y}_j)^2}$$

- R^2 , conocido como coeficiente de determinación, aplicado a la relación entre valores reales y predicciones y la varianza total respecto al promedio.

$$R^2 = 1 - \frac{\sum_{j=1}^n (y_j - \hat{y}_j)^2}{\sum_{j=1}^n (y_j - \bar{y})^2}$$

Para evaluar modelos de **clasificación** se determinan una serie de métricas basándose, primero, en los datos aportados por la denominada *matriz de confusión*, que se construye enfrentando en una tabla, de tantas filas y columnas como número de clases exista en el conjunto de datos, la contabilidad de los casos reales y los predichos, produciéndose cuatro tipos de categorías distintas:

- Verdadero Positivo (True Positive, TP): El valor real es positivo y la prueba predijo que también lo era.
- Verdadero Negativo (True Negative, NP): El valor real es negativo y la prueba predijo lo mismo.
- Falso Positivo (False Positive, FP): El valor real es negativo pero la prueba predijo que era positivo, conocido como *error tipo-I*.
- Falso Negativo (False Negative, FN): El valor real es positivo y la prueba predijo que era negativo, conocido como *error tipo-II*.

En la figura 3.12 se puede observar un ejemplo de esta matriz.



Figura 3.12: Esquema de la matriz de confusión [4]

Una vez contabilizada la matriz anterior, existen una serie de parámetros para validar el desempeño del modelo. Los más reconocidos son:

- *Accuracy*: Medida de exactitud. Para el total de datos, ¿cuantos son ciertos?

$$\text{Accuracy} = \frac{(TP + TN)}{\text{Total}}$$

- *Precision*: Medida de precisión. Para los casos que el modelo ha clasificado como positivos, ¿cuantos son ciertos?

$$\text{Precision} = \frac{TP}{TP + FP}$$

- *Recall*: Medida de sensibilidad. Para los casos reales positivos, ¿cuantos predice?

$$TPR = \frac{TP}{TP + FN}$$

- *Specificity*: Medida de particularidad. Para los casos reales negativos, ¿cuantos predice?

$$\text{Specificity} = \frac{TN}{TN + FP}$$

- *F1 Score*: Puntuación F1. Combina precisión y sensibilidad en uno para tener en cuenta positivos dados como ciertos y reales omitidos.

$$F1_{score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- *False Positive Rate*: Ratio de falsos positivos. El complementario de la particularidad.

$$FPR = 1 - \text{Specificity} = 1 - \frac{TN}{TN + FP} = \frac{(TN + FP) - TN}{TN + FP} = \frac{FP}{TN + FP}$$

Función de coste También llamada función de pérdida, se usa en algoritmos de aprendizaje supervisado para cuantificar como de buenas son las predicciones de un modelo, de manera que esta función calcula un valor que refleja el error o discrepancia entre ambas.

Durante el entrenamiento, el objetivo es siempre **minimizar** dicha función, ajustando los pesos o parámetros del modelo para que sus predicciones sean lo más similares posibles a los valores reales. La elección de la función de coste, y su optimización, depende del tipo de problema y algoritmo empleado.

Aunque existen algoritmos de métodos lineales cuya función de coste permiten una resolución no iterativa y cerrada del mismo, como en el método base de este trabajo FedHEONN, en la mayoría de casos se suele recurrir a métodos numéricos para optimizar dicha función de manera iterativa hasta una esperada convergencia a un mínimo, o al criterio de parada especificado.

Las funciones de pérdida comunes incluyen el *MSE* visto antes, la entropía cruzada (en redes neuronales) o la precisión, y los métodos numéricos de optimización usados varían desde descenso de gradiente estocástico (*SGD Stochastic Gradient Descent*), ADAM (*Adaptive Moment Estimation*), entre muchos otros.

A continuación se ilustra uno de estos procesos de minimización en la figura 3.13.

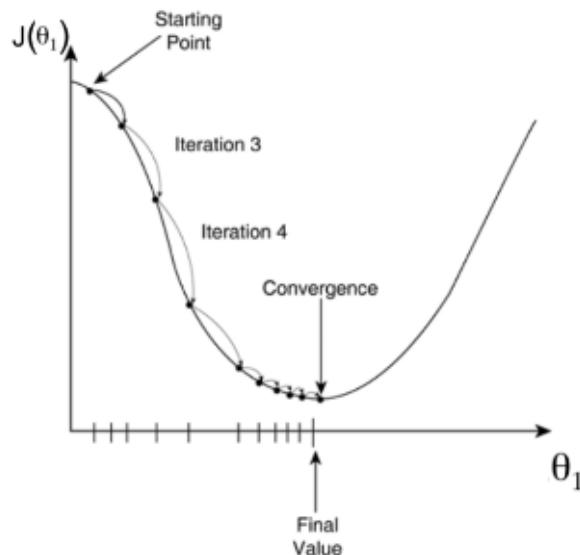


Figura 3.13: Ejemplo de resolución iterativa, por gradiente descendente, de una función de pérdida usada en regresión basada en el MSE [21]

Hiperparámetros Se conocen como hiperparámetros todos aquellos factores que se establecen antes del proceso de entrenamiento del modelo y que no se *aprenden* directamente de los datos. Estos parámetros controlan la configuración, comportamiento y diseño del algoritmo de aprendizaje, y su ajuste es crucial a la hora de obtener un buen rendimiento del modelo.

A diferencia de los pesos o parámetros del modelo, que se aprenden durante su entrenamiento y optimización de la función de pérdida, los hiperparámetros son seleccionados por el usuario, como por ejemplo la tasa de aprendizaje, el número de capas ocultas en una red neuronal artificial (RNA), el tamaño del lote de trabajo, el número máximo de iteraciones, la profundidad de un árbol de decisión, etc.

Existen técnicas para buscar qué hiperparámetros ofrecen un mejor rendimiento del modelo; algunas son brutas y sistemáticas como la búsqueda en cuadrícula (*grid search*) o aleatoria (*random search*) y otras algo más avanzadas como la basada en los algoritmos genéticos. En la figura 3.14 podemos ver un ejemplo de este concepto.

⑥ Despliegue

El despliegue y uso práctico de un modelo de ML implica la integración del modelo final en un entorno de producción, donde podrá generar los resultados esperados y ser utilizado para la toma de decisiones.

Los únicos factores que se tienen que tener en cuenta aquí son la escalabilidad del modelo, su rendimiento, su capacidad de actualización y mantenimiento, la privacidad y seguridad de los datos y las posibles regulaciones e impacto derivados de su uso.

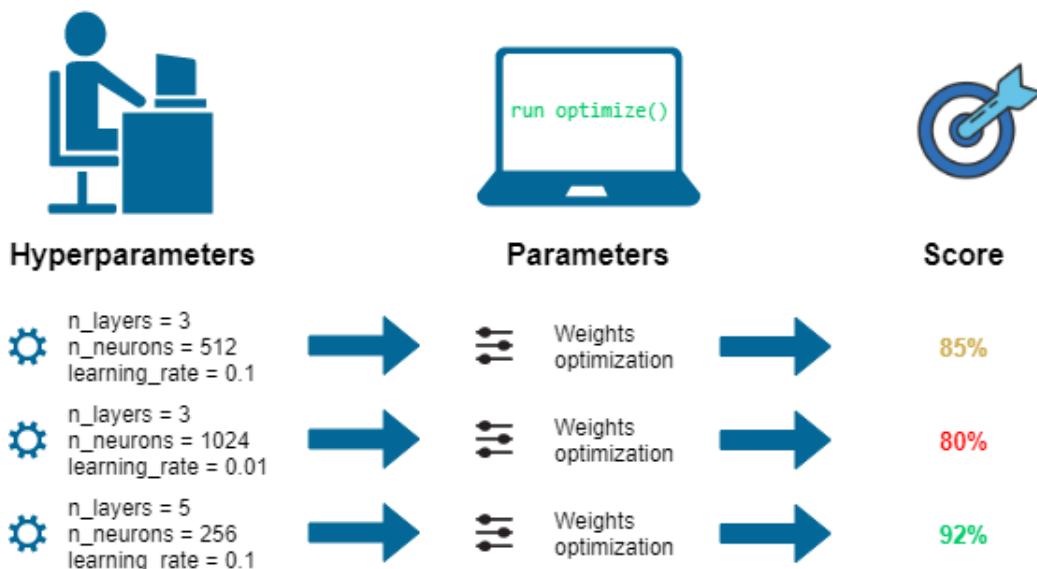


Figura 3.14: Ajuste de hiperparámetros para un modelo de aprendizaje supervisado [18]

3.3.3. Aprendizaje federado

Con el auge de los dispositivos móviles y otros dispositivos embebidos – de uso doméstico o no – englobados en el llamado *Internet of Things* (IoT) se produce, cada vez más, una abundancia de datos que, siendo adecuados y aprovechables para el entrenamiento de modelos de aprendizaje automático, se desaprovechan por restricciones físicas o éticas, debido a su naturaleza privada o sensible, además de encontrarse distribuidos entre millares de dispositivos desconocidos.

Con esta motivación e ideas de fondo fue como surgió el término de *Federated Learning* o Aprendizaje Federado, acuñado por primera vez en el artículo de McMahan et al. [25] titulado '*Communication-Efficient Learning of Deep Networks from Decentralized Data*', en donde se da una primera definición y enfoque a la particular casuística y características que atañen a estos modelos.

En dicho artículo se centran específicamente en el caso de los dispositivos móviles, intentando usar los datos contenidos en estos para usarlos en modelos de ML que mejoren la usabilidad de los mismos; como se materializaría – a posteriori – con su implantación en el teclado móvil *Gboard* del SO *Android* para mejorar la predicción de palabras y *emojis* del usuario.

Presentan así una solución al uso de estos datos sensibles y distribuidos – que imposibilitan el enfoque tradicional, es decir, aunarlos en un centro de datos y realizar con ellos el desarrollo del modelo – defendiendo una nueva técnica de aprendizaje que permite a los usuarios aprovechar colectivamente los beneficios de los modelos sin necesidad de almacenar los datos de forma centralizada ni llegar a compartirlos, creando una red federada de dispositivos que resuelva la función objetivo bajo la coordinación de un servidor central.

Cada uno de los dispositivos – o clientes – mantiene su conjunto de datos intacto de forma local y sólo comparte con el servidor unas actualizaciones iterativas de los pesos de sus respectivos modelos locales, que ni siquiera son necesarios guardar en el servidor central una vez aplicados, en completa sintonía con los principios de *focused collection* y *data-minimization*.

propuestos en el informe de ciberseguridad de la Casa Blanca [30] de 2013 '*Consumer data privacy in a networked world*'.

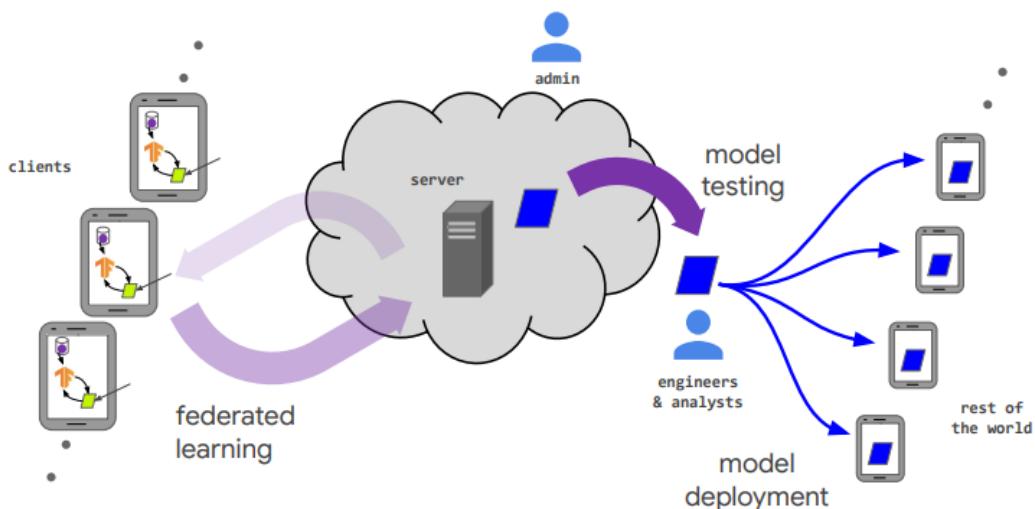


Figura 3.15: Ejemplo de aplicación, ciclo de vida y actores durante el entrenamiento de un sistema de Aprendizaje Federado [17]

En resumen, podemos decir que las técnicas de Aprendizaje Federado surgen como solución a los problemas de aislamiento y fragmentación de datos sensibles que, a mayores, requieren, por motivos éticos y legales, adecuarse a restricciones de privacidad y seguridad.

Definición, características y proceso de desarrollo

Debido a que, en la primera introducción del término *Federated Learning*, se hizo un enfoque especial a su uso en dispositivos móviles, presentamos una definición más general de este concepto extraída del libro de Kairouz et al. [17] de años posteriores, que lo explica como:

El Aprendizaje Federado es un entorno del aprendizaje automático en el que múltiples entidades (clientes) colaboran en la resolución de un problema de aprendizaje automático, bajo la coordinación de un servidor central o proveedor de servicios. Los datos brutos de cada cliente se almacenan localmente y no se intercambian ni transfieren; si no que, para lograr el objetivo del aprendizaje, se utilizan actualizaciones iterativas destinadas a su agregación inmediata.

Caben dos apuntes en la definición anterior; la primera, es que estas 'actualizaciones iterativas' tienen el alcance justo y limitado para contener la mínima información necesaria para la tarea de aprendizaje en cuestión – realizando su agregación lo antes posible evitando su almacenaje y posible reconstrucción – y, la segunda, es que no se tiene en cuenta las técnicas de aprendizaje federado totalmente descentralizadas (P2P), que carecen de servidor central y difieren de la arquitectura clásica cliente-servidor de la definición, como ilustra la figura 3.16.

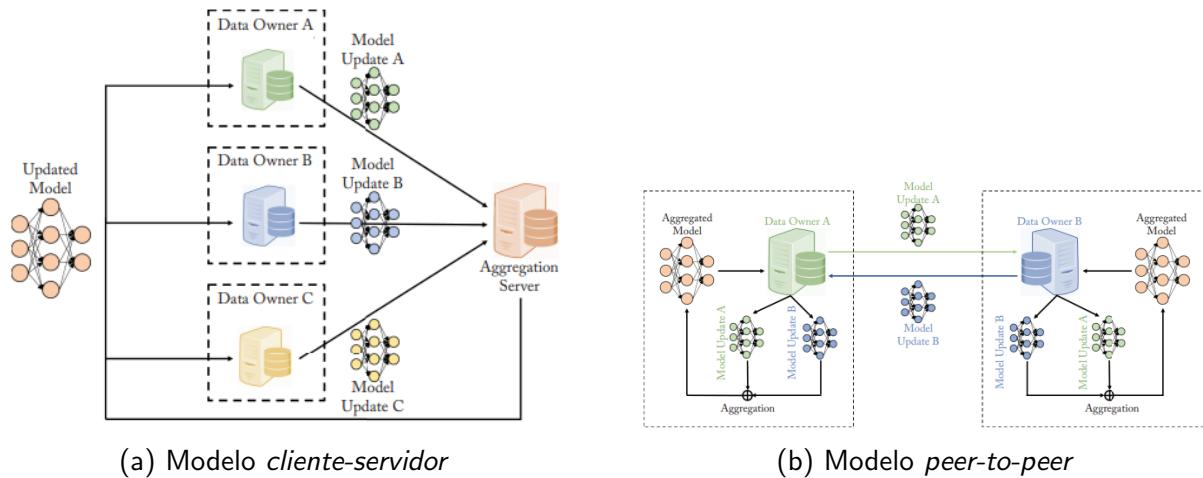


Figura 3.16: Arquitecturas en sistemas de Aprendizaje Federado [34]

De uno u otro modo se mantienen los datos de entrenamiento descentralizados, mitigando muchos de los riesgos y costes asociados a la privacidad y derivados del aprendizaje automático tradicional y centralizado que, además, como veremos en futuros capítulos, es menos eficiente energéticamente.

Entre las **características de los problemas de optimización** a los que se enfrentan las técnicas de aprendizaje federado se encuentran, normalmente, las siguientes:

- **Distribución estadísticas desiguales y no uniformes:** en estos entornos, no se pueden asumir los supuestos de las técnicas tradicionales de ML distribuido, como son el de conjuntos de datos *IID* (Independent and Identically Distributed), es decir, muestras independientes entre sí y con idénticas distribuciones de probabilidad. Y es que hay que suponer que, aunque cada cliente aporte datos locales correspondientes al mismo objetivo, debido al uso y circunstancias de cada uno, su distribución ha de ser distinta.
- **Desequilibrado de datos:** Por la misma razón que el punto anterior, no se puede suponer que las muestras han de tener tamaños iguales o que contengan el mismo número de datos sobre clases o *clusters* que el resto.
- **Escalabilidad masiva:** debido a que los datos pueden estar distribuidos sobre un gran número dispositivos, es requisito fundamental que estos sean capaces de manejar distribuciones masivas de clientes.
- **Diferencias computacionales:** Debido a la diversidad de dispositivos que pueden conformar estas redes federadas, se ha de tener en cuenta distintas capacidades computacionales, que pueden afectar, dependiendo del hardware disponible, en el número de iteraciones o rondas en las que participe el cliente o el tiempo esperado de respuesta de cada uno de ellos.
- **Comunicaciones costosas:** A diferencia de lo que sucede en escenarios clásicos de computación distribuida, donde existe un gran ancho de banda bajo redes locales, en las redes federadas resulta ser lo contrario, con clientes que no siempre pueden estar disponibles y que usan conexiones lentas y poco fiables, por lo que el modelo tiene que poder gestionar desconexiones e incorporaciones súbitas.

Una vez conocidos los principales retos a los que se enfrenta el *problema de la optimización*

federada, podemos introducir las fases típicas del **proceso de entrenamiento** de un modelo de aprendizaje federado:

1. **Selección de clientes**: El coordinador central o servidor escoge un subconjunto de clientes que cumplan los requisitos apropiados.
2. **Difusión**: Los clientes seleccionados obtienen el estado del modelo global actual y su programa de entrenamiento.
3. **Cálculos locales**: Cada dispositivo realiza la computación del programa de entrenamiento sobre su conjunto de datos, por ejemplo SGD en un lote local de datos (para un caso de *FedAvg*).
4. **Agregación**: El servidor recibe un cúmulo de actualizaciones locales de cada uno de los dispositivos. Los rezagados pueden ser abandonados y no tenidos en cuenta pasado un tiempo máximo. Sobre los disponibles se realiza un promedio ponderado de los pesos obtenidos, según el número de muestras de cada cliente (para un caso de *FedAvg*).
5. **Actualización del modelo global**: Por último, el servidor realiza la actualización del modelo global basándose en la agregación anterior.

Cabe decir que, en el flujo anterior, el servidor/coordinador orquesta todo el proceso de entrenamiento (arquitectura cliente-servidor) y el proceso se repite (en caso de ser un modelo iterativo) durante sucesivas rondas hasta que este es detenido debido a su convergencia, número máximo de rondas o agotado el tiempo máximo de entrenamiento.

Categorías y tipos de aprendizaje federado

Existen distintos tipos de categorizaciones presentes en las técnicas de aprendizaje federado, dependiendo del ámbito al que refieran, y que explicaremos brevemente en esta sección.

En primer lugar, en referencia al tipo y número de clientes que conforman la red federada, se puede hablar de:

- **Cross-silo**: reservado para las aplicaciones de FL que involucren a un número reducido de clientes fiables, como por ejemplo, la colaboración entre múltiples organizaciones o entidades en aras de desarrollar un modelo de ML conjunto.
- **Cross-device**: el apropiado para el resto de casos, donde el énfasis está en redes masivas de dispositivos móviles o embebidos (*edge devices*).

Por último, basado en la estructura de cada conjunto de datos que participe en la red federada, se habla de:

- **Horizontal Federated Learning, HFL**: el *Aprendizaje Federado Horizontal* representa el caso en el que los conjuntos de datos de cada uno de los clientes comparten una estructura de atributos semejante, como se observa en la figura 3.17. De este modo, los participantes tienen *datasets* similares en términos de características pero que no pueden ser fusionados debido a restricciones de privacidad o seguridad. Al colaborar y combinar sus modelos locales se logra un modelo global más robusto y generalizado sin revelar los datos sensibles subyacentes.

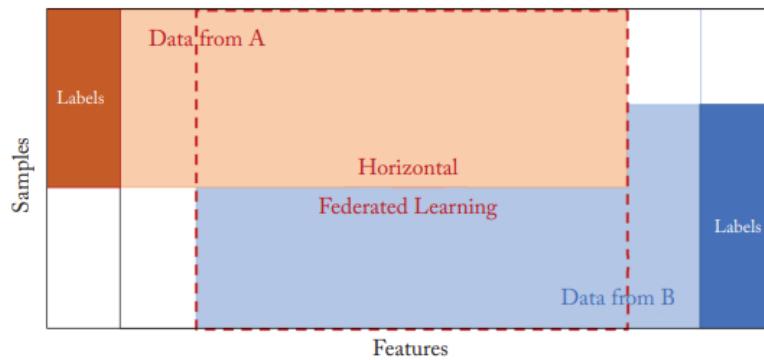


Figura 3.17: Estructura de datos compartidas en HFL [34]

- **Vertical Federated Learning, VFL:** el *Aprendizaje Federado Vertical* se centra en la colaboración entre múltiples participantes o clientes que poseen características diferentes pero complementarias en sus respectivos conjuntos de datos. Como mínimo, deben tener en común algún tipo de identificador que permita alinear los datos de los distintos conjuntos implicados mediante un proceso criptográfico conocido como *encrypted-entity-alignment*. Este escenario es ideal para la colaboración entre múltiples entidades que comparten un grupo de clientes pero donde cada una de ellas tiene información diferente sobre cada uno de ellos. Así, podrían cooperar en el desarrollo de un modelo conjunto que beneficie a ambos sin revelar datos confidenciales de sus clientes el uno al otro.

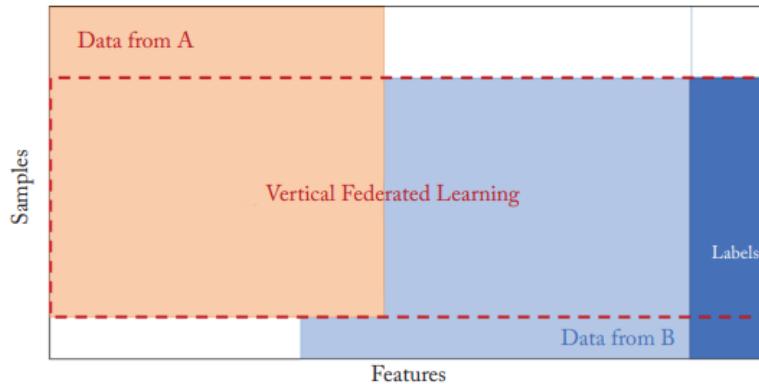


Figura 3.18: Estructura de datos complementarios en VFL [34]

Optimización

En esta sección explicaremos, en detalle, uno de los algoritmos de optimización más utilizados en las técnicas de aprendizaje federado y que, a su vez, fue el propuesto inicialmente en el artículo de McMahan et al [25] mencionado al principio del capítulo. Se trata del denominado *FedAvg*, o promedio federado.

Este algoritmo de optimización establece un sistema de actualizaciones síncronas que avanzan en consecutivas rondas de comunicación cliente-servidor. Asume un número fijo de K clientes, cada uno con un conjunto de datos local invariable. Al principio de cada ronda, una fracción aleatoria C de participantes es seleccionada, y el servidor envía el actual estado global del modelo a cada uno de los clientes (p.ej. pesos de la RNA). Cada cliente realiza entonces

el entrenamiento de este aplicado sobre su propio conjunto de datos y envía los resultados de vuelta al coordinador. El servidor agrega estos cambios y el proceso se repite.

Este algoritmo es aplicable a cualquier función objetivo de suma-finita de la siguiente forma, siendo \mathbf{w} el vector de parámetros o pesos del modelo y n el número de datos total:

$$\min_{\mathbf{w} \in \mathbb{R}^d} f(\mathbf{w}) \quad \text{donde} \quad f(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{w}) \quad (3.1)$$

Para los problemas de aprendizaje automático, normalmente $f_i(\mathbf{w}) = \ell(x_i, y_i; \mathbf{w})$, que es la pérdida en la predicción realizada por (x_i, y_i) con los parámetros del modelo \mathbf{w}_i . Asumiendo datos particionados entre K clientes, con \mathcal{P}_k la serie de índices de las muestras del cliente k y $n_k = |\mathcal{P}_k|$, podemos reescribir la ecuación (3.1) como:

$$f(\mathbf{w}) = \sum_{k=1}^K \frac{n_k}{n} F_k(\mathbf{w}) \quad \text{donde} \quad F_k(\mathbf{w}) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} f_i(\mathbf{w}) \quad (3.2)$$

Si la partición \mathcal{P}_k se forma distribuyendo uniformemente los datos de entrenamiento entre los clientes de manera aleatoria entonces tendríamos que $\mathbb{E}_{\mathcal{P}_k}[F_k(\mathbf{w})] = f(\mathbf{w})$.

Este supuesto uniformemente distribuido es lo que se conoce como condición **IID**, y se asume típicamente en algoritmos de optimización distribuidos, aunque, como ya vimos, este algoritmo FedAvg no contempla este caso; es decir, F_k podría ser una mala aproximación aleatoria de f , por lo que se trabaja bajo condiciones **no IID**.

Bajo este tipo de funciones de coste se ha aplicado satisfactoriamente, en aprendizaje automático distribuido, distintas variantes del llamado *Gradiente Descendente Estocástico* o SGD para su optimización, del que **FedAvg no es si no otra generalización de este**.

El **SGD** se podría aplicar directamente al problema de optimización federada, donde solo se realice, por ronda, un único cálculo de gradiente por lote para aquellos clientes seleccionados; sin embargo, este enfoque es eficiente computacionalmente pero muy costoso en comunicaciones, debido al gran número de rondas necesarias para producir buenos resultados. Por lo tanto, se apostará por realizar múltiples iteraciones (*epochs*) locales con el objetivo de avanzar el aprendizaje y reducir el número total de rondas globales, disminuyendo las rondas de comunicación necesarias para la convergencia, que es el verdadero cuello de botella de estos escenarios. Así, tomando como base el SGD, el algoritmo **FedAvg** lo generaliza añadiendo tres parámetros:

- **C : fracción de clientes seleccionados cada ronda.**
- **E : número de iteraciones de entrenamiento realizadas en cada cliente sobre su conjunto de datos local.**
- **B : tamaño de los lotes de datos locales usados para el entrenamiento.**

Cabe mencionar que si $C = 1$ $E = 1$ $B = \infty$ entonces FedAvg se convierte sin más en la aplicación de SGD en una red federada, denominada **FedSVD**.

Cuando se añaden más iteraciones locales de estos cálculos de gradiente por cada cliente ($E \uparrow$) se generaliza, finalmente, a **FedAvg**, cuyo algoritmo queda esquematizado en el pseudocódigo 1.

Algoritmo 1 *FederatedAveraging*. Los K clientes se indexan por k , B es el tamaño de lote local, E el número de iteraciones (*epochs*) locales y η la tasa de aprendizaje.

```

1: Coordinador o servidor central:
2: inicializa  $w_0$ 
3: for cada ronda  $t = 1, 2, \dots$  do
4:    $m \leftarrow \max(C \cdot K, 1)$ 
5:    $S_t \leftarrow$  (selección aleatoria de  $m$  clientes)
6:   for cada cliente  $k \in S_t$  en paralelo do
7:      $w_{t+1}^k \leftarrow$  ActualizarCliente( $k, w_t$ )
8:   end for
9:    $m_t \leftarrow \sum_{k \in S_t} n_k$ 
10:   $w_{t+1} \leftarrow \sum_{k \in S_t} \frac{n_k}{m_t} w_{t+1}^k$ 
11: end for
12:
13: ActualizarCliente( $k, w$ ):
14:    $\mathcal{B} \leftarrow$  (Particionar  $\mathcal{P}_k$  en lotes de tamaño  $B$ )
15:   for cada epoch local  $i = 1, \dots, E$  do
16:     for cada lote  $b \in \mathcal{B}$  do
17:        $w \leftarrow w - \eta \nabla \ell(w; b)$ 
18:     end for
19:   end for
20:   return  $w$  al servidor

```

Amenazas y privacidad de datos

Aunque el atractivo propio de las técnicas de FL es que tienen, **por diseño, una naturaleza privada**, haciendo que los clientes participantes no revelen sus datos sensibles, si no solo actualizaciones de los modelos locales que entran – cumpliendo así la filosofía del *data minimization* –, esto no quita que sigan siendo vulnerables a cierto tipo de ataques.

Podemos distinguir dos escenarios de posibles amenazas:

- **Global**: encargado de proteger el funcionamiento del sistema federado frente a dispositivos externos maliciosos u *honestos pero curiosos*.
- **Local**: encargado de evitar fugas de información sensible del cliente hacia el servidor durante las comunicaciones que realizan entre sí.

Aunque la privacidad no se puede tratar como una cualidad binaria o perfectamente cuantificable, si se puede presentar una serie de objetivos de privacidad para justificar el uso de distintas herramientas y tecnologías en ciertas partes del flujo de trabajo.

Existen tres aspectos de la privacidad de estos sistemas que tienen que ser comprendidos y analizados para saber qué tipo de remedio se puede usar para reforzarlos. Podemos imaginar y simplificar el proceso de las técnicas de aprendizaje federado como el de la evaluación de una función f en un conjunto de datos distribuidos en varios clientes. Dada esta analogía, habría que tener en cuenta:

1. Como se calcula f y cuál es el flujo de información de los resultados intermedios, ya que estos son susceptibles a un cliente, servidor o administrador malicioso. Además de

la privacidad por diseño del propio FL (*data minimization*), aquí se introducen técnicas de computación segura multi-parte (Secure Multiparty Computation, **SMP**C).

2. Que es lo que se calcula, es decir, cuanta información sobre un participante se revela a terceros o a los usuarios finales por el resultado del despliegue de f . Para contrarrestarlo se aplican métodos de privacidad diferencial (Differential Privacy, **DP**).
3. Por último resta la verificabilidad, entendida como la capacidad de un cliente o servidor de probar a otros participantes del sistema que han ejecutado el entrenamiento o programa al que se han comprometido fielmente, sin revelar los datos potencialmente sensibles sobre los que actuaron. Para esto, se hace uso de técnicas conocidas como pruebas de conocimiento cero (Zero-Knowledge-Proofs, **ZKP**).

A continuación explicamos brevemente algunas de estas técnicas y cuál es la escogida en el algoritmo de partida de este trabajo.

Secure Multi-Party Computation

También conocidos como protocolos seguros o computación segura distribuida, se trata de un campo de la criptografía encargado de hacer que una red de participantes calculen una función acordada, con sus datos locales como entradas, de tal manera que solo revelen la salida de estas a los demás clientes, sin que se pueda inferir de estos resultados ninguna información acerca de las primeras.

Una de estas técnicas es la llamada **encriptación homomórfica** (*Homomorphic Encryption*, HE), que permite que ciertas operaciones matemáticas – normalmente suma y multiplicación – se realicen directamente sobre textos cifrados sin tener que descifrarlos primero, produciendo como resultado otro texto cifrado que contiene la salida correspondiente. La figura 3.19 ilustra este concepto. En el escenario de FL, esto permite a clientes y servidor realizar el agregado de las actualizaciones recibidas del modelo manteniendo los valores de los pesos/parámetros ocultos.

Existen distintos tipos de encriptación homomórfica, dependiendo del tipo de operaciones matemáticas que permitan realizar sobre los datos cifrados. Por lo general, las implementaciones más computacionalmente eficientes no son capaces de evaluar funciones u operaciones complejas, por lo que hay que llegar a un equilibrio entre requisitos y recursos:

- FHE, o *Fully Homomorphic Encryption*: permite realizar cálculos como sumas y multiplicaciones y evaluar funciones arbitrariamente complejas.
- PHE, o *Partially Homomorphic Encryption*: permite realizar solo un subconjunto acordado de operaciones matemáticas, como por ejemplo suma o multiplicación, pero sin mezclar ambas.

Un punto flaco de la implementación de estas técnicas en sistemas de aprendizaje federado es la cuestión de quien alberga las claves privadas del esquema de encriptación, ya que este actor tendría que ser totalmente confiable, o bien reregar esta a un tercero no participante acordado.

En este trabajo, como veremos más adelante, partimos de un modelo que implementa FHE como medio para asegurar la privacidad y protección entre ambas partes del entorno federado.

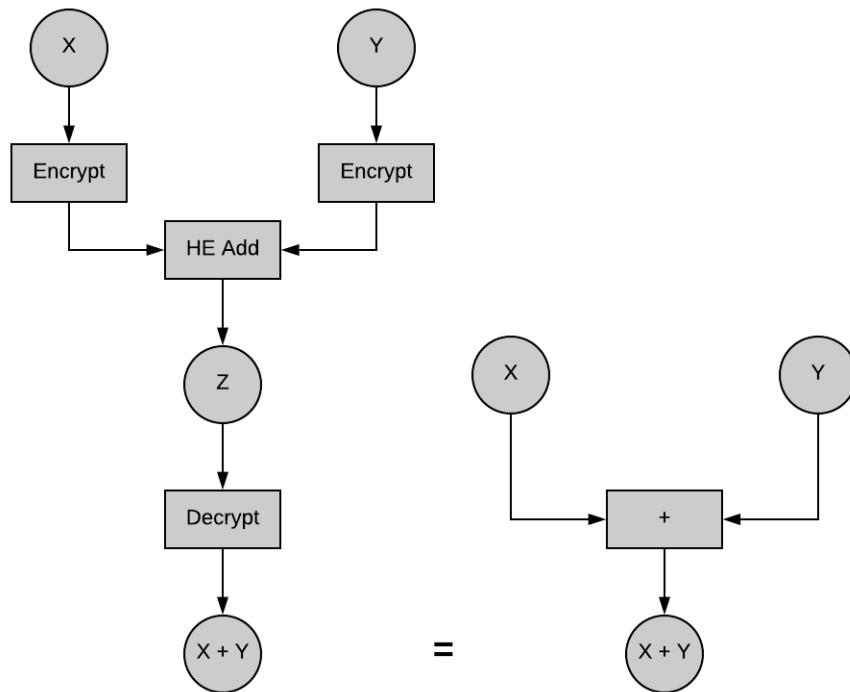


Figura 3.19: Comparativa de suma bajo esquema FHE y sin él [5]

Differential Privacy

La privacidad diferencial se entiende como una herramienta para cuantificar y limitar la fuga de información sensible sobre los participantes, de modo que pretende introducir cierto nivel de incertidumbre en el modelo, suficiente para enmascarar la contribución de cualquier usuario individual. La DP se cuantifica mediante los parámetros de pérdida de privacidad (ϵ, δ) , donde menor (ϵ, δ) corresponde a mayor privacidad.

Normalmente en FL, el papel de coordinador es la parte confiada a implementar el mecanismo de DP en el sistema, asegurándose que sólo salidas 'privatizadas' del modelo son liberadas a terceros o al usuario final. Sin embargo, se puede reducir la necesidad de esta parte de confianza usando *privacidad diferencial local*, si cada participante perturbase su propio modelo añadiendo ruido antes de la agregación de actualizaciones, por ejemplo.

Zero Knowledge Proof

Las pruebas de conocimiento cero son unas técnicas criptográficas que permiten a un cliente probar a otro que ha ejecutado cierto computo o programa sobre unos datos según lo acordado, sin comprometer ni revelar ninguna información sensible sobre ella.

En el contexto de FL se puede usar para dos situaciones: el servidor podría demostrar que ha realizado ciertas operaciones específicas como la agregación o adición de ruido para DP y, por último, los clientes podrían demostrar al servidor que sus entradas y comportamiento han seguido la especificación o protocolo acordado.

Casos de uso, software y datasets

Siendo un mecanismo que permite modelos de ML personalizados y compartidos, construidos sobre conjuntos de datos descentralizados y esparcidos entre múltiples participantes sin comprometer la privacidad y seguridad de estos tiene, por tanto, un futuro prometedor en numerosos ámbitos, como el financiero, sanitario, educativo, *smart city* o *edge computing*, incentivado por el número creciente de dispositivos móviles y embebidos conectados a internet (*edge devices*, dentro del marco conocido como *IoT*).

En cuanto al *software* disponible a día de hoy, en el ámbito de la investigación y simulación, algunas de las librerías más destacadas son:

- *TensorFlow Federated*: framework de código abierto de FL y otros cálculos con datos descentralizados para la investigación y experimentación simulada, en Python.
- *PySyft*: biblioteca en Python de código abierto que añade seguridad y privacidad, desacoplando los datos del entrenamiento del modelo. De los creadores de TenSEAL.
- *Flower*: framework que provee infraestructura para el aprendizaje y evaluación de algoritmos de FL, para los cuales es agnóstico, facilitando la implementación y el uso de diseños propios.
- *PyVertical*: proyecto para la implementación de aprendizaje federado vertical (VFL) en Python, usando PySyft.

En cuanto a los conjuntos de datos para el entrenamiento de estos modelos, se tienen que simular conjuntos descentralizados, normalmente diseccionando reconocidos conjuntos centralizados.

3.3.4. Algoritmos verdes

La Inteligencia Artificial y, por tanto, el Aprendizaje Automático, no están exentos de traer consigo riesgos para el cambio climático derivados del coste energético asociado al desarrollo y aplicación de estas tecnologías. En este sentido, parece que jueguen roles opuestos ya que, por un lado, pueden colaborar como papel fundamental en la reducción de esta crisis – al diseñar redes eléctricas inteligentes (*smart-grid*), desarrollando infraestructura sostenible o modelando nuevos sistemas climáticos – y, por otro, se están convirtiendo en un emisor de carbono cada vez más significante.

Por ejemplo, en el artículo '*Energy and Policy Considerations for Deep Learning in NLP*' [33], de 2019, se cuantificaron los costes medioambientales y el CO_2 equivalente del entrenamiento de modelos avanzados de procesamiento de lenguaje natural (NLP), estimando que cada uno supuso alrededor de 300.000 kg de dióxido de carbono, parejo a las emisiones producidas por 125 vuelos de ida y vuelta entre *Nueva York* y *Beijing*, para hacerse una idea.

Pero es que, además de esto, en el anterior estudio sólo se tuvieron en cuenta los costes de entrenamiento de dicho modelo basándose puramente en su demanda computacional aunque, una visión más acertada y global de este impacto, debería de considerar también el efecto de la infraestructura necesaria en torno al despliegue de estos sistemas por parte de las *Big Tech*, así como el origen de la energía dispensada a estos. Este tema se tratará en más detalle en el apartado de emisiones y huella de carbono de la IA de esta misma sección.

Es por eso que, cada vez más, se está apostando por una concienciación para el desarrollo de la IA dentro de un contexto de responsabilidad, comprendiendo el impacto medioambiental que suponen y fomentando el desarrollo de una inteligencia artificial sostenible, que ponga en equilibrio **eficiencia** y **eficacia** y aplique buenas prácticas, marcando una distinción entre lo que algunos artículos ya denominan **Algoritmos Verdes** [19] o **Green & Red AI** [31].

En la última sección, para finalizar, hablaremos de algunas de las iniciativas políticas puestas en marcha a nivel europeo y nacional para fijar marcos de referencia, objetivos de desarrollo sostenible e integración en el tejido productivo de estas nuevas tecnologías.

En definitiva, es importante tener en cuenta que, aunque la IA es una herramienta con muchísimo potencial, también se puede convertir en una gran consumidora de recursos, por lo que hay que tener presente que los beneficios – progreso científico, tecnológico, social, etc. – de esta deberá siempre compensar sus inconvenientes.

IA verde y roja

Para asegurarse de que la IA sea beneficiosa también en el aspecto medioambiental es importante alentar a los investigadores a que prioricen el empleo y desarrollo de *hardware* y algoritmos computacionalmente eficientes, así como estandarizar la publicación de informes de rendimiento que contengan resultados de tiempos de entrenamiento y susceptibilidad de los hiperparámetros del modelo.

Y es que, en un estudio realizado por *Dario Amodei* [3], en 2018, se descubrió que los costes computacionales utilizados para el entrenamiento de grandes modelos de IA se vienen duplicando desde 2012 cada tres meses y medio aproximadamente, como se observa en la figura 3.20. De continuar esta tendencia, significaría un incremento incluso mayor que el observado en la clásica *Ley de Moore* sobre el número de transistores en los circuitos integrados.

Otro artículo, titulado '*Green AI*' [31], ha acuñado el término **IA Verde** para describir la tendencia reciente de desarrollar el aprendizaje automático de manera sostenible, en contraste con lo que también denominan **IA Roja**.

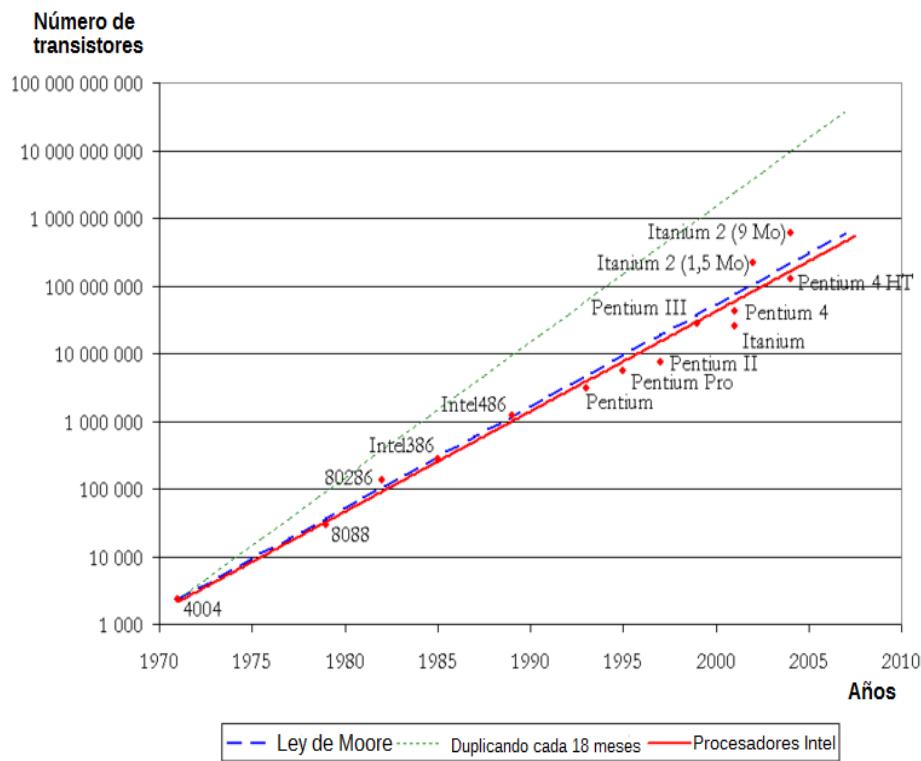
En este estudio se hace un llamado a considerar la **eficiencia** como un criterio de igual importancia – al mismo nivel que las métricas de eficacia – en las investigaciones y desarrollos de ML. Además, propone la inclusión de un precio o coste financiero que refleje los gastos asociados al desarrollo y entrenamiento de estos modelos, premiando así el diseño y la eficiencia y no la fuerza bruta.

No por ello hay que restar importancia a los resultados obtenidos por dichos modelos, ya que contribuyen y llevan al extremo las fronteras conocidas de la IA, pero el retorno en las mejoras de rendimiento obtenidas se van a ver reducidas cada vez más, al tiempo que demandan mayores recursos.

Existen tres factores determinantes para entender lo que se considera como **IA Roja**, que serían:

1. **E**: coste de ejecutar el modelo con una sola muestra.
 2. **D**: tamaño del *dataset* o conjunto de datos.
 3. **H**: número de ajustes en hiperparámetros, es decir, cuantas veces el modelo es entrenado durante su desarrollo.
- R**, coste de producción de un determinado resultado o rendimiento.

$$\text{Coste}(R) \propto E \cdot D \cdot H \quad (3.3)$$



(a) Ley de Moore comparada con datos históricos [32]

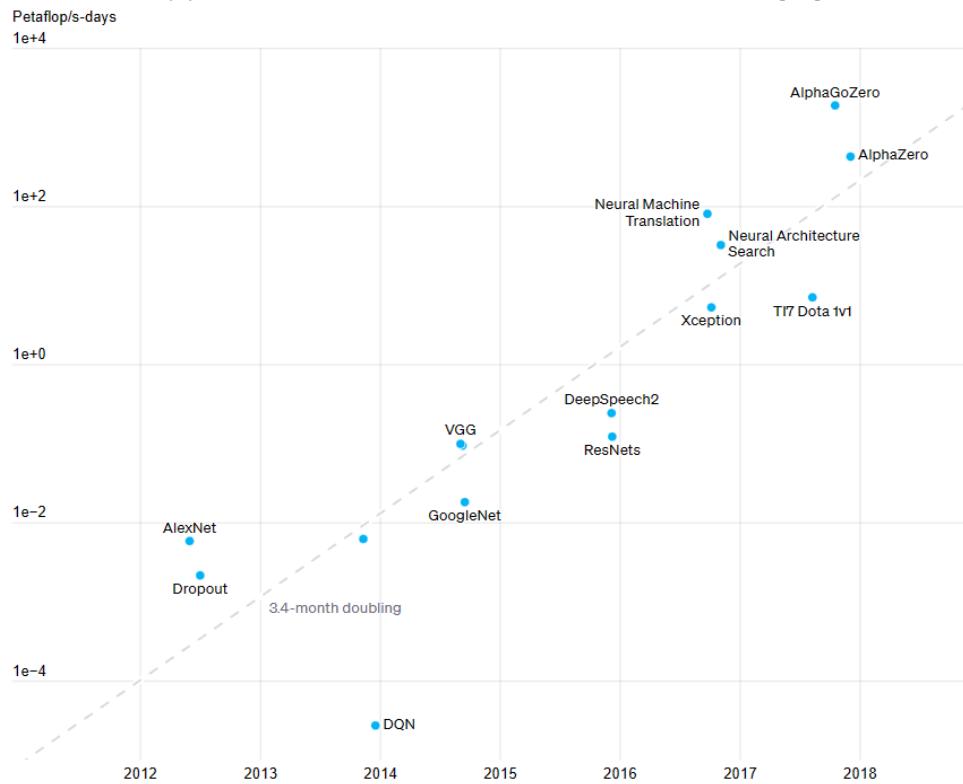
(b) Coste computacional, en $pfs \cdot \text{día}^2$, para el entrenamiento de distintas IA [3]

Figura 3.20: Comparación entre ratios de crecimiento del número de transistores en circuitos integrados y costes computacionales en IA³

Cuando los costes de producir un resultado en un modelo de IA incrementan linealmente con alguno de los factores anteriores, estamos ante IA Roja. Contrariamente, definen la **IA Verde** como:

Investigación en inteligencia artificial que produzca nuevos resultados sin aumentar su coste computacional, e idealmente reduciéndolo.

En el citado artículo [31] también se dan una serie de recomendaciones en cuanto a métricas de eficiencia a utilizar en las investigaciones de IA, entre las cuales destacamos las siguientes:

- **Consumo eléctrico**
- **Tiempo de ejecución**
- **Número de parámetros**
- **FPO** (Floating Points Operations)

Las tres primeras métricas son agnósticas en cuanto a la fuente de energía utilizada pero dependientes del *hardware* sobre el que se ejecute el modelo, por lo que puede ser difícil distinguir las contribuciones de un mejor diseño a las de una mejor maquina. Por ello el estudio se decanta por utilizar el **FPO**, es decir, el número de operaciones de coma flotante como estimación de la cantidad de trabajo realizado. Esta métrica está ligada directamente al consumo energético y al tiempo de ejecución, sin depender del *hardware* subyacente.

En la figura 3.21 se puede observar esta tendencia en la disminución de la mejora de la exactitud con el sucesivo aumento de los costes computacionales.

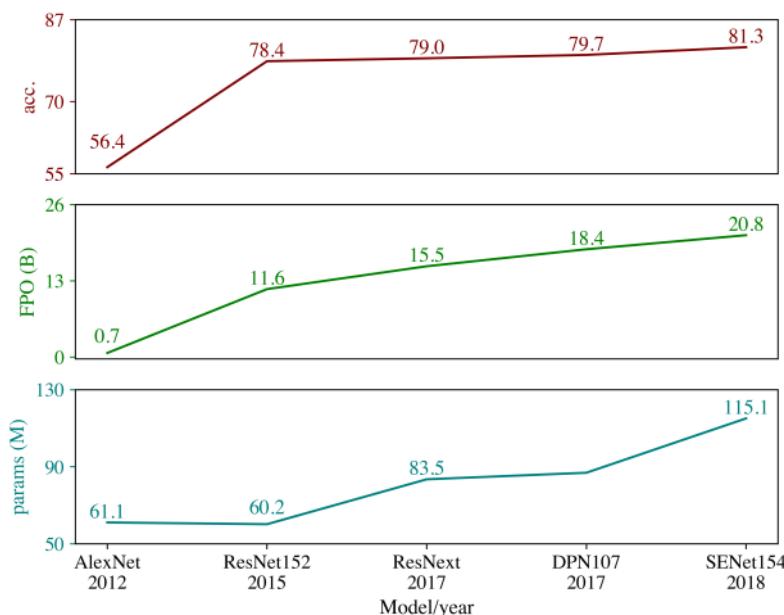


Figura 3.21: Distintos modelos de detección de objetos en visión artificial: exactitud (acc) - FPO (billones) y número de parámetros del modelo (millones) [31]

²petaflop/s·día ($pfs \cdot \text{día}$) recordemos que los FLOPS son operaciones de coma flotante por segundo y el $pfs \cdot \text{día}$ intenta dar una medida de consumo computacional semejante al $kW \cdot h$ para energía eléctrica.

³Advertir como ambos ejes verticales, para el número de transistores y de $pfs \cdot \text{día}$, se encuentran en escala logarítmica.

No todos los problemas requieren soluciones basadas en el ML, y resolver problemas sencillos que pueden ser tratados con una IA menos intensiva computacionalmente, con redes neuronales profundas, por ejemplo, es un despilfarro.

Emisiones y huella de carbono de la IA

Una métrica que no abordó el estudio anterior es el de las emisiones de carbono e impacto medioambiental, considerada fuera del alcance al depender estas de la infraestructura y fuentes de energía locales que abastecan a los equipos, resultando inviable comparar investigaciones realizadas en lugares y épocas distintas.

Y es que, por ejemplo, en un dossier reciente publicado por el *Electric Power Research Institute* (EPRI) [2], se da cuenta de como el consumo eléctrico de los centros de datos de las denominadas *Big Tech* se ha doblado en el período 2017-2021, y como esperan llegar a suponer hasta el 9.1 % del consumo eléctrico total de Estados Unidos para el año 2030, comparado con el 4 % estimado a fecha de 2024. Además, esta demanda energética se halla geográficamente concentrada, lo cual complica todavía más la carga eléctrica que deben soportar estas redes. Este aumento viene impulsado por la adopción masiva de los modelos generativos de IA, por ejemplo *ChatGPT*, del cual estima dicho estudio requerir 2.9 W·h por consulta, diez veces más que una búsqueda en *Google*, con 0.3 W·h.

Y no es solo el consumo energético; en recientes publicaciones [20], se da cuenta del grandísimo consumo de agua requerido por estos *data centers*. Así, el entrenamiento del modelo *GPT-3* requirió de 700.000 litros de agua corriente para refrigerar los equipos de estos centros de datos. Estiman, además, que la demanda global de IA para 2027 requerirá de unos 5000 millones de m^3 de agua, que, en comparación, supone la mitad de la consumida anualmente en todo Reino Unido.

Es por ello que cada vez aparecen más herramientas para contabilizar estos aspectos, como las presentadas a continuación. En ambas se trabajan formas de cuantificar las emisiones de carbono teniendo en cuenta la ubicación de la maquina que ejecuta el entrenamiento del modelo, su duración, el tipo de energía que usa y el *hardware* en el que se realiza. La primera es una especie de simulador web y la segunda herramienta se puede integrar fácilmente dentro del propio código del modelo.

Cálculo y simuladores

En el estudio '*Green Algorithms: Quantifying the carbon footprint of computation*' [19], realizaron un simulador web para calcular estas emisiones, llamado Green Algorithms, como se observa en la figura 3.22:

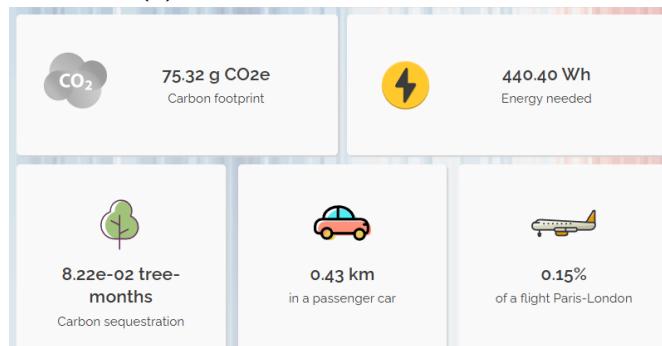
Los cálculos realizados para producir esas estimaciones se computan así:

$$E = t \cdot (\eta_c \cdot P_c \cdot \mu_c + \eta_m \cdot P_m) \cdot PUE \cdot 0.001 \quad (3.4)$$

Donde, en (3.4), t es el tiempo de ejecución en horas, η_c el número de núcleos del CPU, η_m el tamaño de la memoria RAM disponible en GB, μ_c es el factor de uso del CPU, P_c la potencia requerida por cada uno de los núcleos en W, P_m la potencia requerida por la memoria en W y, por último, *PUE* (*Power Usage Effectiveness*) la eficiencia del centro de datos, medida como la razón entre la potencia o energía requerida total dividido entre la destinada al uso del equipamiento TI.

The screenshot shows the 'Details about your algorithm' section of the Green Algorithms website. It includes fields for Runtime (HH:MM), Type of cores (CPU), Number of cores (4), Model (Core i7-9700K), Memory available (in GB) (16), Select the platform used for the computations (Personal computer), Select location (Europe, Spain), and two questions about usage factors with radio button options and numerical inputs (0.8 and 1).

(a) Parámetros de la herramienta



(b) Resultados de la herramienta

Figura 3.22: Simulador web para la estimación del cálculo de CO_2 equivalente de una RNA [19]

Por último, la estimación de la huella de carbono ($g CO_2^{eq}$) la hacen multiplicando la energía precisada en (3.4) ($kW \cdot h$) multiplicada por el CI , *carbon intensity* ($\frac{gCO_2^{eq}}{kW \cdot h}$), de la localización:

$$C = E \cdot CI \quad (3.5)$$

Por otro lado, el equipo de *CodeCarbon* presenta un planteamiento muy similar al anterior, con la diferencia de que han desarrollado una librería en Python que se puede integrar en cualquier algoritmo desarrollado con este lenguaje de programación y que, monitorizando directamente los recursos disponibles y el *hardware* de la maquina, realiza las estimaciones de las ecuaciones (3.4) y (3.5) automáticamente, generando los informes correspondientes, como los que se aprecian en la figura 3.23.

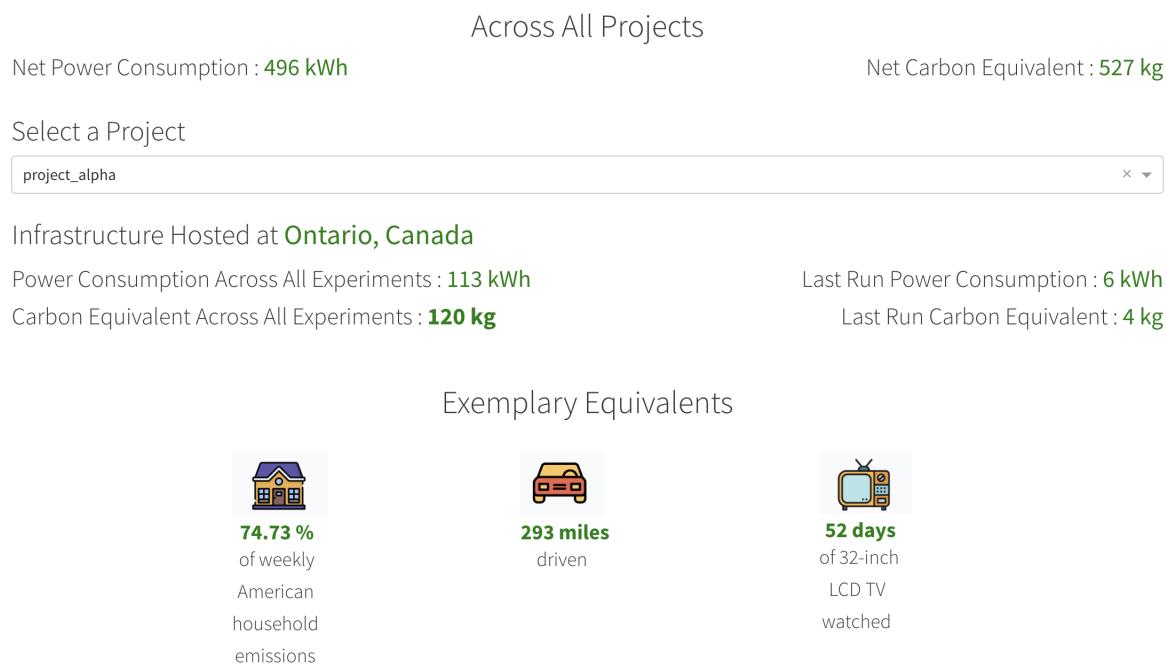


Figura 3.23: Informe generado por la herramienta CodeCarbon [7]

Iniciativas políticas europeas y nacionales

Existe cada vez mayor interés, por parte de entidades gubernamentales a nivel europeo y nacional, de introducir marcos regulatorios, buenas prácticas y fomentar el desarrollo sostenible en temas relacionados con la computación de altas prestaciones como la IA u otras tecnologías intensivas tipo *blockchain* como respuesta al conflicto climático, reconociéndolas como fuentes de emisiones en cuanto atañe a las tecnologías de la información.

Así tenemos, por ejemplo, el amplio estudio promocionado por la Comisión Europea '*The role of Artificial Intelligence in the European Green Deal*' [28], donde se exponen una serie de recomendaciones para equilibrar un buen uso de estas herramientas con su impacto medioambiental e integrarlos en las cadenas de valor y tejido industrial. Respecto a la ejecución de algoritmos, la UE está estudiando también mejores prácticas para instaurar un mercado de tecnologías de computación en la nube respetuosas con el medioambiente.¹

En el panorama nacional existen dos iniciativas que surgen del *Ministerio de Asuntos Económicos y Transformación Digital*, en concreto de la *Agenda España Digital 2026*, que son el ENIA [10] – *Estrategia Nacional de Inteligencia Artificial* – y el PNAV [8] – *Programa Nacional de Algoritmos Verdes* –, esta última incluida como expansión de uno de los ejes estratégicos de la primera, y cuyos puntos clave y objetivos desarrollamos a continuación:

1. **Fomentar la investigación en materia de inteligencia artificial verde**, a través de financiación pública.
2. **Promover el uso de infraestructuras y servicios eficientes**, apoyando el desarrollo de estándares y herramientas para la medición del consumo de estos algoritmos e incentivando el consumo energético renovable y sostenible de los centros de datos que los

¹*Energy-efficient Cloud Computing Technologies and Policies for an Eco-friendly Cloud Market*

ejecuten.

3. **Integrar la inteligencia artificial verde y Blockchain en el tejido productivo**, creando nuevos mercados económicos.
4. **Dinamizar el mercado español a través de la inteligencia artificial verde**, promocionando su adopción en sectores que contribuyan al cambio climático.

Notación Big-O

Una herramienta esencial a la hora de describir la complejidad computacional de un algoritmo es la llamada Notación Big-O, conocida también como notación *Asintótica* o *Landau*, en honor a uno de los autores a los que se atribuye, Edmund Landau, matemático alemán de comienzos del siglo pasado.

La Notación Big-O se usa para medir como de bien **escala** un algoritmo o programa – según aumenta el número de elementos procesados – respecto al tiempo que tarda en ejecutarlo, es decir, nos da el *ratio de crecimiento* de dichas operaciones.

Es una medida muy útil para comparar la **eficiencia** entre algoritmos, ya que aunque no nos proporcione el tiempo de ejecución exacto de estos, sí nos da una idea de cómo de rápido se incrementa este tiempo de ejecución en función del número de datos de entrada. Así, se clasifican los algoritmos según la clase de ratio de crecimiento que tengan, una medida de su buena/mala escalabilidad.

En la tabla 3.1 se resumen algunas de estas clases de notaciones y sus nombres, y en la figura 3.24 se puede observar gráficamente la representación de estas, enfrentando el número de pasos o operaciones requeridas por el algoritmo frente al número de elementos.

Notación	Nombre
$O(1)$	constante
$O(\log n)$	logarítmico
$O(n)$	lineal
$O(n \cdot \log n)$	lineal logarítmico
$O(n^2)$	cuadrático
$O(2^n)$	exponencial
$O(n!)$	factorial

Tabla 3.1: Órdenes más utilizados en análisis de algoritmos para la notación Big-O

Mayor número de operaciones representa mayor tiempo de ejecución y mayor coste computacional, lo cual implica un mayor coste energético y por lo tanto una **menor eficiencia energética**.

Veamos, con un ejemplo, las tres propiedades que se suelen asumir a la hora de analizar el grado O de una función $f(n)$:

1. Si una función o algoritmo se puede escribir como una suma finita de otras funciones, entonces marca el grado de $f(n)$ la que tenga mayor ratio de crecimiento.
2. Si una función o algoritmo es multiplicada por algún factor constante, es decir, uno que no dependa de n , se puede omitir.

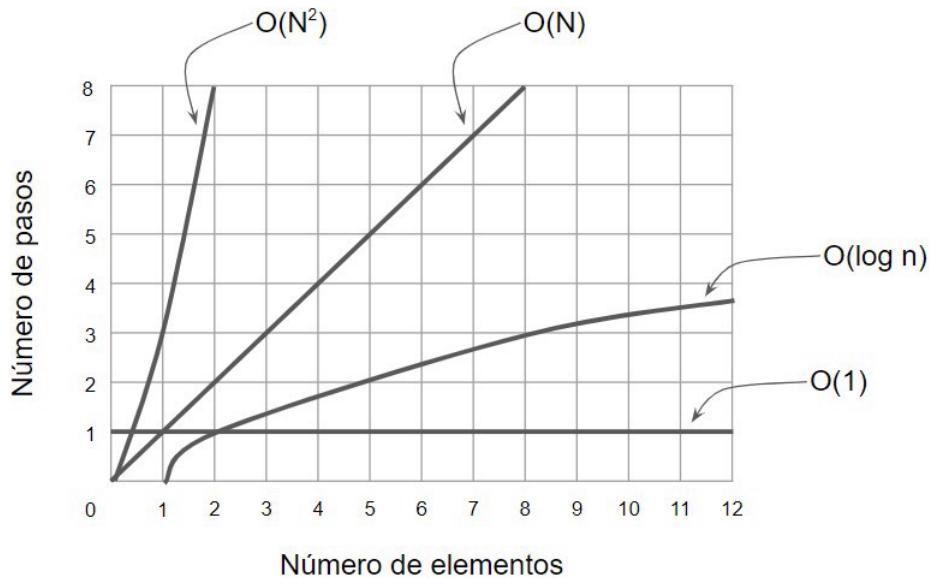


Figura 3.24: Gráfica del ratio de crecimiento y escalabilidad de las distintas clases de notaciones Big-O [1]

3. Si un algoritmo es producto de dos funciones el grado resultante del mismo es el producto de ambos.

$$f_1 = O(g_1) \quad f_2 = O(g_2) \rightarrow f_1 f_2 = O(g_1 g_2) \quad (3.6)$$

Dadas las anteriores características, es fácil demostrar que, para el siguiente ejemplo (3.7), el grado de complejidad de la función es cúbico:

$$f(n) = 5 \log n + 5(\log n)^4 + 2n^2 + n^3 = O(n^3) \quad (3.7)$$

Asegurarse de que los algoritmos de optimización presentes en los modelos usados en FL sean lo más eficientes posibles implica un análisis de este tipo para saber como se comporta ante distintos volúmenes de datos de entrada, y saber así como de bien escala. Obviamente es preferible algoritmos de ordenes bajos, como lineal o logarítmico, pero eso dependerá únicamente del modelo y las técnicas de optimización empleadas.

Existe, sin embargo, una razón por la que el aprendizaje federado se beneficia y gana mayor eficiencia que sus contrapartidas tradicionales, aún usando modelos y algoritmos de optimización equivalentes, y que se pueden explicar usando esta notación:

1. Sea un modelo de ML \mathcal{M} cuya fase de entrenamiento \mathcal{E} implica un cuello de botella en su proceso de optimización marcado por el uso de cierto algoritmo de optimización \mathcal{A} clasificado como de tiempo cuadrático $O(n^2)$.
2. Sea θ el nº de operaciones, τ el tiempo de ejecución y ϵ el coste energético, de modo que:

$$\uparrow \theta \rightarrow \uparrow \tau \rightsquigarrow \uparrow \epsilon$$

3. Si el volumen de datos de entrada para cada ronda de \mathcal{E} es de η , esto implica que:

$$\mathcal{A}_{ML}(\eta) \rightsquigarrow O(\eta^2) \implies \theta_{ML}$$

4. Si este mismo volumen es distribuido entre múltiples participantes – p.ej. **cuatro** – de una red federada, las fases de \mathcal{E}_{local} se benefician de una reducción inmensa en sus tiempos de ejecución y coste energético:

$$\mathcal{A}_{fed}(\eta) = \mathcal{A}_{local}\left(\frac{\eta}{4}\right) \cdot 4_{clientes} \rightsquigarrow O\left(\left(\frac{\eta}{4}\right)^2\right) \cdot 4_{clientes} \implies \frac{\theta}{4} \rightarrow \frac{\tau}{4} \rightarrow \frac{\epsilon}{4} \quad (3.8)$$

5. Es decir, una reducción de tres cuartos en el tiempo, coste y construcción del modelo, siempre que supongamos condiciones ideales y una distribución perfectamente paralela y síncrona de clientes con igual poder computacional.

He aquí uno de los grandes beneficios de estas técnicas de **FL**. Aún suponiendo participantes heterogéneos, con distintas potencias computacionales, y siendo estas, por separado, ínfimas en comparación a la que pueda tener una máquina o servidor de un gran centro de datos, se pueden encontrar beneficios derivados de su diseño distribuido tanto por tiempos de entrenamiento como en eficiencia energética, debido a que cada cliente entrena un modelo con los datos locales disponibles que luego se agregan en un modelo global.

3.4. Diseño e implementación

En este capítulo se describe el diseño e implementación del sistema informático sobre el que se realizarán las pruebas de los distintos métodos desarrollados. Empezaremos con una vista global del conjunto y de su arquitectura, para luego describir en detalle los dispositivos embebidos, sistemas operativos, lenguajes de programación y librerías empleadas.

3.4.1. Vista global del sistema

Contamos con cinco dispositivos físicos a nuestra disposición para el desarrollo de las pruebas e investigación: dos ordenadores portátiles, dos modelos diferentes de Raspberry Pi y una Jetson Nano.

La división de trabajo y almacenamiento de datos se hará acorde a los límites y características de cada uno de ellos. Por esta razón los portátiles, al tener mayor capacidad, serán los encargados de albergar el servidor así como sucesivos lotes de clientes simulados, cuando realicemos las pruebas de escalabilidad.

El método desarrollado es una aplicación de un sistema de aprendizaje automático federado, en donde la computación y entrenamiento del modelo se hace de manera distribuida. Así, se conforma una red compuesta de varios dispositivos interconectados, siendo uno de ellos coordinador del proceso global, rol asignado al ordenador portátil más potente, trabajando en conjunto. También se han realizado pruebas, por comparativa, con algoritmos y técnicas equivalentes con arquitectura centralizada, llevadas a cabo de igual modo en dicho portátil.

Arquitecturas

El diseño de estos sistemas se suele hacer siguiendo una arquitectura **cliente-servidor** clásica, donde un servidor orquestará el proceso de selección, entrenamiento, agregación y validación del modelo, y unos clientes que, teniendo acceso cada uno a su propio conjunto de datos, entrenarán un modelo local y recibirán las actualizaciones y directrices necesarias cuando sean llamados. Existe otro tipo de arquitecturas federadas, como, por ejemplo, las *Peer-To-Peer* (P2P), que no hemos cubierto en este trabajo debido a que no ha sido la empleada en nuestro modelo. A continuación detallamos el funcionamiento de la arquitectura escogida:

Cliente-Servidor Este tipo de arquitectura es la más utilizada en sistemas federados, conocida también como diseño tipo *master-worker*. En ella, K participantes o clientes, cada uno con un subconjunto del grupo de datos, entranan de manera colaborativa el modelo de aprendizaje automático con la ayuda de un servidor – también denominado coordinador o servidor de agregación –. Los supuestos para estos casos es que los participantes son honestos mientras que el servidor es honesto pero curioso (*honest-but-curious*). Este concepto implica que, aunque el servidor no altere maliciosamente los resultados o cómputos del proceso, el diseño del algoritmo de entrenamiento tiene que prevenir cualquier tipo de fuga de información útil que este pueda recibir de los clientes.

El proceso de entrenamiento de estos sistemas consiste normalmente de los siguientes pasos.

1. Los clientes entran localmente sus modelos y enmascaran sus resultados – p.ej. gradientes – mediante técnicas criptográficas o privacidad diferencial, para luego enviarlos de vuelta al servidor.
2. El servidor recopila los resultados y realiza su agregación.
3. El servidor envía de vuelta los resultados agregados a los participantes.
4. Los clientes actualizan sus respectivos modelos locales con los resultados recibidos desencriptados.

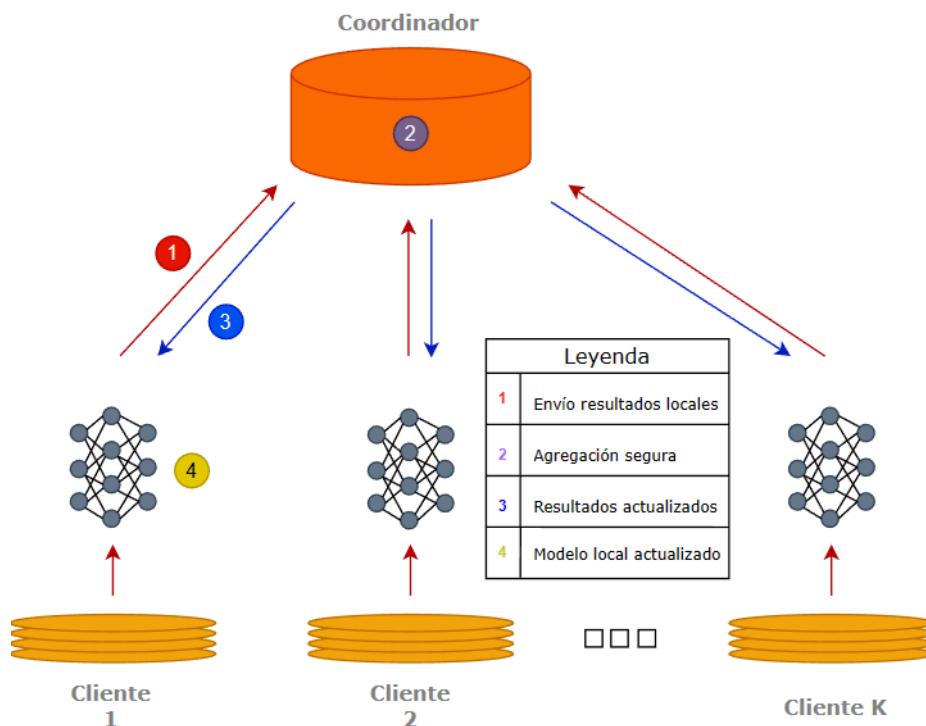


Figura 3.25: Ejemplo de arquitectura cliente-servidor para sistema federado

En el caso de tratarse de un algoritmo iterativo, el proceso anterior se repite hasta que la función objetivo (*loss function*) converja o se alcance el número máximo de rondas posibles.

3.4.2. Sistemas informáticos

Introducimos brevemente las especificaciones técnicas de los dispositivos físicos utilizados en el trabajo.

Sistemas embebidos

Se denominan así a los sistemas de computación basados en microprocesadores o microcontroladores que están diseñados para realizar unas pocas funciones dedicadas y limitadas; en contraste con los ordenadores personales, de propósito general.

Raspberry Pi Son una serie de computadoras de placa única (*SBC*), de bajo coste y tamaño reducido, creadas en el Reino Unido por la *Raspberry Pi Foundation* para fomentar el aprendizaje de informática y electrónica. Lanzada en 2012, ha evolucionado con sucesivas versiones,

cada una con diferentes especificaciones. Estas placas se utilizan en proyectos variados como domótica, servidores web, centros multimedia, robótica y todo el ámbito del *IoT*.

Todas sus versiones incluyen un procesador Broadcom, memoria RAM, puertos USB, HDMI, GPU, Ethernet, 40 pines GPIO, conector para cámara y ranura para tarjetas SD, lo que permite ejecutar sistemas operativos y aplicaciones de software. Su sistema operativo oficial es *Raspberry Pi OS*, basado en Debian, aunque también es compatible con otros SO basados en arquitectura ARM.

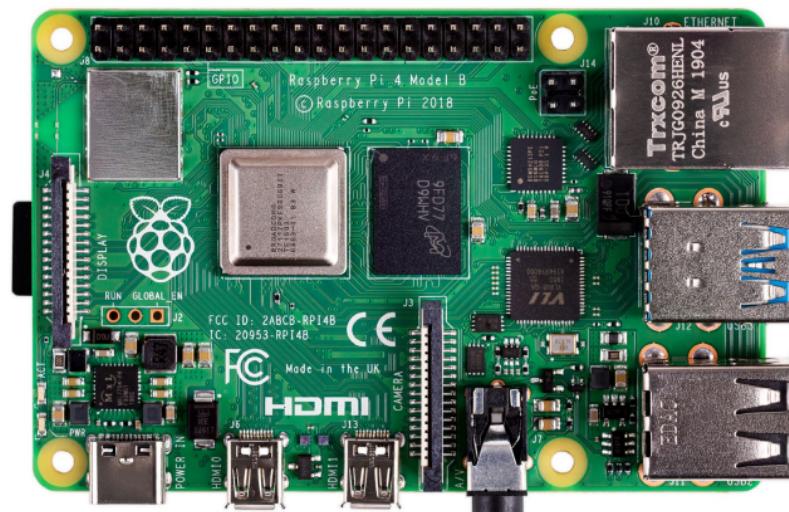


Figura 3.26: Raspberry Pi Model 4B

En la figura 3.26 se puede apreciar el aspecto de uno de estos modelos, el denominado Raspberry Pi Model 4B. Junto con otro, más pequeño y de menor rendimiento, llamado Raspberry Pi Zero W, que se puede observar en la figura 3.27, son los dos dispositivos embebidos de este fabricante que emplearemos en este Trabajo Final de Grado.

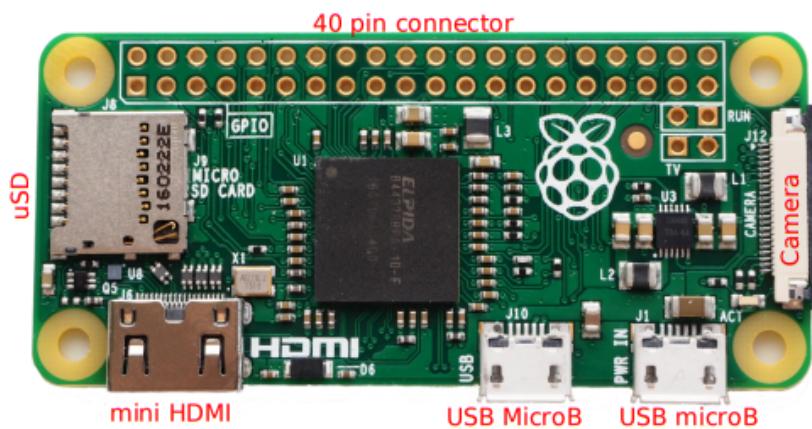


Figura 3.27: Raspberry Pi Zero W

En las tablas 3.2 y 3.3 presentamos las especificaciones de cada uno de ellos:

Procesador	Broadcom BCM2711 quad-core Cortex-A72 (ARM) @ 1.5GHz
Memoria	1GB/2GB/4GB/8GB LPDDR4 (dependiendo del modelo).
Conectividad	Wireless LAN. Bluetooth 5.0. BLE. Gigabit Ethernet. 4×USB.
GPIO	Encabezado GPIO estándar de 40-pines.
Video y Audio	2×microHDMI. MIPI DSI. MIPI CSI.
Multimedia	VideoCore VI @ 500 MHz GPU. Gráficos OpenGL ES 3.0.
Tarjetas SD	Ranura para MicroSD para SO y almacenamiento.
Alimentación	5V DC via USB-C. 5V DC via GPIO. PoE (requiere PoE HAT).
Entorno	Temperatura de funcionamiento 0-50°C.

Tabla 3.2: Especificaciones de la Raspberry Pi Model 4B

Procesador	ARM 11 Broadcom BCM2835 single-core CPU @ 1GHz
Memoria	512MB RAM.
Conectividad	Wireless LAN. Bluetooth 4.1. BLE. 2×microUSB.
GPIO	Encabezado GPIO estándar de 40-pines.
Video y Audio	miniHDMI. MIPI CSI.
Multimedia	VideoCore IV @ 400MHz GPU. Gráficos OpenGL ES 2.0.
Tarjetas SD	Ranura para microSD para SO y almacenamiento.
Alimentación	5V DC via microUSB. 5V DC via GPIO.
Entorno	Temperatura de funcionamiento 0-50°C.

Tabla 3.3: Especificaciones de la Raspberry Pi Model Zero W

Jetson Nano Son una serie de dispositivos embebidos de la compañía Nvidia que están diseñados específicamente para aplicaciones de IA y robótica. Sus modelos son pequeños, de bajo consumo y ofrecen un alto rendimiento para las tareas de procesamiento de datos intensivas de estos campos, como procesamiento de imágenes, detección de objetos, reconocimiento de voz o aprendizaje profundo. Se ofrecen diversos modelos, que vienen equipados con un amplio rango de potentes procesadores CPU/GPU, incluyendo además interfaces para cámaras, sensores y otros dispositivos externos.

En este Trabajo Final de Grado se ha utilizado el modelo Jetson Nano B01, del que se puede apreciar su aspecto en la figura 3.28 y sus especificaciones en la tabla 3.4.

Procesador	Quad-core Arm A57 processor @ 1.43 GHz.
Memoria	4GB LPDDR4 RAM.
Conectividad	Ethernet. 4×USB. 1×microUSB.
GPIO	260-pin SO-DIMM. Puertos expansión GPIO 40-pines.
Video y Audio	HDMI. eDP 1.4. 4×MIPI CSI.
Multimedia	128-core Maxwell GPU.
Tarjetas SD	Ranura para microSD para SO y almacenamiento.
Alimentación	5V/4A DC via cable. 5V/2A via microUSB. PoE.
Entorno	Temperatura de funcionamiento -25/80°C.

Tabla 3.4: Especificaciones de la Jetson Nano



Figura 3.28: Jetson Nano

Sistemas físicos y simulados

Los ordenadores portátiles son Lenovo, modelos Legion Y540-15IRH y ThinkPad X200, cuyo aspecto y especificaciones se pueden observar en la figura 3.29 y en las tablas 3.5 y 3.6, respectivamente.



(a) Lenovo Legion Y540



(b) Lenovo ThinkPad X200

Figura 3.29: Ordenadores portátiles disponibles

Con respecto a los sistemas simulados, nos referimos a como, ante la falta de dispositivos físicos suficientes, se simularon multitud de clientes, generados artificialmente mediante *software* bajo uno de estos dos equipos, para realizar las pruebas de escalabilidad del entorno federado.

Procesador	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
Memoria	16GB RAM.
Conectividad	Ethernet. Wireless LAN. Bluetooth. 3×USB.
Video y Audio	HDMI. DP. USB-C.
Multimedia	GTX 1660 Ti.
Almacenamiento	1TB SSD NVMe.
Alimentación	20V/11.5A DC via cable.
Entorno	Temperatura de funcionamiento -30/100°C.

Tabla 3.5: Especificaciones del Lenovo Legion Y540

Procesador	Intel(R) Core 2 Duo (TM) P8600 CPU @ 2.40GHz
Memoria	2 GB RAM.
Conectividad	Ethernet. Wireless LAN. Bluetooth. 3×USB.
Video y Audio	VGA.
Multimedia	Intel GMA 4500MHD.
Almacenamiento	250 GB SSD.
Alimentación	20V/3.25A DC via cable.
Entorno	Temperatura de funcionamiento -30/100°C.

Tabla 3.6: Especificaciones del Lenovo ThinkPad X200

3.4.3. Sistemas operativos

Los sistemas operativos escogidos para los diferentes dispositivos físicos se resumen en la tabla 3.7.

Dispositivo	Sistema Operativo
Lenovo Legion Y540	Windows 10 (64-bit)
Raspberry Pi Model 4B	Raspberry Pi OS Lite (32-bit), basado en Debian
Raspberry Pi Model Zero W	Raspberry Pi OS Lite (32-bit), basado en Debian
Jetson Nano B01	Jetson Nano Developer Kit Image (Linux4Tegra)

Tabla 3.7: Sistemas operativos utilizados

3.4.4. Lenguajes de programación y librerías

El lenguaje de programación utilizado durante todo este Trabajo Final de Grado ha sido **Python**, en concreto su versión 3.10, por las siguientes razones:

- Facilidad de uso, configuración e instalación sencilla.
- Gran abanico de librerías y herramientas para los campos de IA y ML.
- Multiplataforma y gratuito.
- El modelo base de este estudio estaba implementado en este lenguaje.

En cuanto a las bibliotecas empleadas durante el desarrollo de este trabajo, destacamos las siguientes:

- **Numpy**: facilita las operaciones con matrices multidimensionales, proporcionando herramientas para la manipulación de datos y cálculo numérico.
- **Pandas**: utilizada para la manipulación y análisis de datos en estructuras conocidas como *DataFrames*, con forma de tablas.
- **Scipy**: librería que contiene diversos módulos de optimización, álgebra lineal, integración e interpolación, entre otros.
- **Scikit-learn**: amplia biblioteca empleada para el aprendizaje automático y la minería de datos, que incluye varios algoritmos de clasificación, regresión, análisis de grupos, reducción de dimensionalidad y preprocesamiento de datos, por nombrar algunos.
- **TenSEAL**: librería criptográfica que proporciona una API para Python capaz de realizar encriptación homomórfica sobre vectores y matrices.
- **CodeCarbon**: ayuda a medir y rastrear la huella de carbono generada por la ejecución de código en proyectos de ciencia de datos y aprendizaje automático.
- **FastAPI**: framework para construir APIs web, de tipo REST, de manera rápida y eficiente, diseñado para ser fácil de usar.

Un detalle pormenorizado de todas las librerías empleadas se puede encontrar en el repositorio enlazado en la sección 4.2.

3.5. Metodología

En esta sección exponemos resumidamente los conjuntos de datos escogidos, el preprocesamiento aplicado, las métricas empleadas, la validación de los modelos y la medición de su consumo energético realizado durante todo el transcurso de las pruebas y experimentos de los siguientes apartados.

Conjuntos de datos

La mayoría de *datasets* empleados en este trabajo son de clasificación, tanto binaria como multiclase; el único conjunto de datos de regresión usado fue el llamado *Carbon Nanotubes*. Las principales características de cada uno se resumen en la tabla 3.8.

Conjunto de datos	Muestras	Atributos	Clases/Salidas	Clasificación
Skin	245,057	3	2	✓
Dry Bean	13,611	16	7	✓
MiniBooNE	130,065	50	2	✓
MNIST-orig	70,000	784	10	✓
MNIST-simp	5,620	64	10	✓
MNIST-test	1,797	64	10	✓
Carbon Nanotubes	10,721	3	3	X

Tabla 3.8: Características de los conjuntos de datos

Cabe destacar que hemos planteado 3 versiones distintas del conocido *dataset* y *benchmark* MNIST. Como se trata de un conjunto bastante voluminoso, aprovechamos unas versiones reducidas y preprocesadas, denominadas *MNIST-simp* y *MNIST-test*, que están disponibles en el repositorio del UCI ML Repository y en la documentación de Scikit-Learn, respectivamente, para el prototipado y ajuste de hiperparámetros, mientras que dejamos el *dataset* original para validaciones y posterior experimentación.

Procesamiento de datos

Todos los *datasets* pasan por el mismo preprocesado antes de ser alimentados a los modelos. Primero, se realiza una división en dos conjuntos, uno de **entrenamiento**, con el 70 % de los datos totales, y otro de **test**, con el 30 % restante. Luego, **normalizamos** ambos conjuntos usando una técnica conocida como *StandardScaling*, resultando en distribuciones con media cero y desviación estándar unidad (*gaussiana*).

Por último, se implementa la posibilidad de reordenar los conjuntos de clasificación en distribuciones independientes e idénticamente distribuidas (**IID**), mezclando aleatoriamente el conjunto de datos, o en distribuciones no IID, ordenando secuencialmente las muestras según sus clases. Esto nos permitirá estudiar la versatilidad del modelo, del que deberíamos ser capaces de obtener los mismos resultados independientemente de si los clientes entrenaen distribuciones IID o no IID.

El tratamiento de valores faltantes o la detección y eliminación de *outliers* no fue necesario, puesto que estos *datasets* ya vienen 'curados'.

Métricas empleadas

Para los conjuntos de clasificación nos hemos centrado en la exactitud o *accuracy* como principal métrica de **eficacia**, mientras que para el caso de regresión hemos usado el error cuadrático medio o *mean squared error* (MSE).

Aún así, implementamos un módulo, denominado `metrics.py`, con diversos algoritmos para calcular otras métricas de utilidad.

Validación y ajuste de modelos

Una vez desarrollado una mejora o variación al algoritmo del modelo de base, es necesario comprobar que las nuevas métricas arrojadas por este son verdaderamente representativas de dicho cambio y no fruto de una casualidad o aleatoriedad propia del conjunto de datos o partición con el que se esté trabajando.

Para ello implementamos en distintos experimentos la **validación cruzada**, mediante la popular técnica conocida como *K-fold*, usando 10 partes o *splits*, lo cual asegura una rotación en los datos de validación y permite sacar medias y desviaciones estándar de las métricas citadas en el apartado anterior.

Por último, una vez corroborado los resultados anteriores, se produce al ajuste de hiperparámetros mediante la técnica exhaustiva conocida como *grid-search*, que prueba todas las posibles combinaciones de un conjunto de hiperparámetros dado.

Con el mejor posible modelo ya ajustado, procedemos a calcular las métricas finales sobre el conjunto de *test*, que contiene los datos aún no utilizados en las fases de entrenamiento y validación del algoritmo.

Análisis energético

El análisis energético y de impacto medioambiental de un modelo previamente contrastado se realizó mediante la integración de la herramienta *CodeCarbon* [7], incluido solo en las fases de entrenamiento y predicción. Por recomendación de los autores de esta herramienta se ha instalado también el programa *Intel Power Gadget 3.6*, para una mejor monitorización de los consumos CPU del proceso.

Aparte de obtener la demanda energética de nuestro modelo en diferentes configuraciones o *setups*, también se han contrastado otro tipo de algoritmos de ML de uso común, principalmente pequeñas redes neuronales artificiales (RNA), para obtener **eficiencias** relativas.

3.6. Análisis de soluciones

En esta sección explicaremos en profundidad el modelo y algoritmo de aprendizaje automático de partida de este trabajo, conocido como *FedHEONN*, así como las mejoras y variaciones realizadas sobre él en este TFG, finalizando con la implementación del servidor y cliente federado prototipo y su análisis energético.

3.6.1. FedHEONN

En el artículo publicado por Fontenla-Romero et al. [12] se presenta el modelo **FedHEONN** como un nuevo método de aprendizaje federado diseñado para mejorar la privacidad y eficiencia en tareas de ML, que implementa encriptación homomórfica (HE) para evitar fugas de información y posibles ataques inversos. Se basa en una red neuronal de una sola capa (sin capas ocultas) que, aún sencilla y limitada, es capaz de resolver multitud de tareas, su entrenamiento es rápido y puede manejar conjuntos de datos voluminosos de manera eficiente.

Arquitectura y función objetivo

En las redes neuronales de una sola capa, el problema de optimización de los parámetros (llamados *pesos óptimos*) que minimizan la función objetivo – normalmente el error cuadrático medio (MSE) – suele ser un problema iterativo debido a la no linealidad introducida por las funciones de activación de la capa de salida; y resuelto, normalmente, por algún algoritmo de optimización como, por ejemplo, *descenso del gradiente estocástico* (SGD), entre otros.

En el caso de FedHEONN, sin embargo, el enfoque principal no está en medir el MSE a la salida de la red, es decir, comparando salidas reales (\mathbf{d}) con las predichas (\mathbf{y}), si no haciéndolo justo antes de pasar por la función de activación, siendo posible así obtener una función objetivo **convexa** y de solución **cerrada**. A continuación presentamos esta idea de modo formal y mostramos la arquitectura y el esquema del MSE medido antes de la función de activación en las figuras 3.30a y 3.30b.

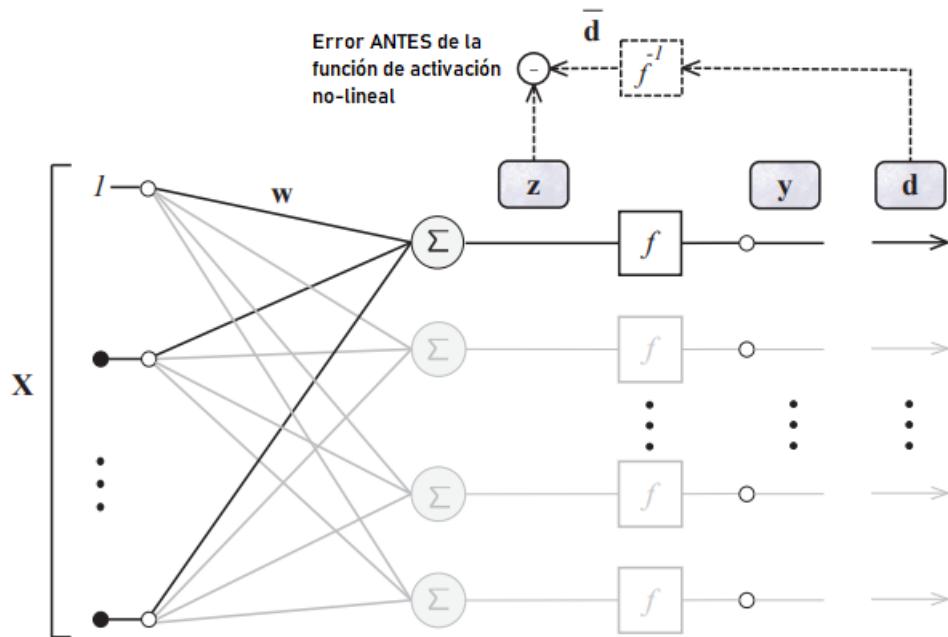
Consideremos primero el caso de un conjunto de datos de entrenamiento centralizado representado por la matriz $\mathbf{X} \in \mathbb{R}^{m \times n}$, siendo m el numero de atributos (incluyendo el *bias* o término descentrado) y n el número de muestras.

Vamos a considerar la matriz de salidas reales o *deseadas*, definida por $\mathbf{d} \in \mathbb{R}^{n \times c}$, donde c es el número de salidas; en lo que sigue, como de única salida (es decir, $\mathbf{d} \in \mathbb{R}^{n \times 1}$) para mayor simplicidad en los cálculos, aunque la extensión al caso general es evidente, ya que cada salida de la red neuronal incumbe a un conjunto independiente de parámetros o *pesos*.

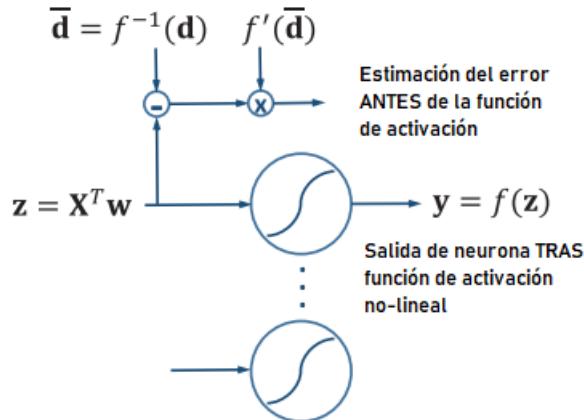
Los pesos estarán así definidos por un vector (que incluye el término descentrado o *bias*), $\mathbf{w} \in \mathbb{R}^{m \times 1}$, de modo que la salida predicha por el modelo (\mathbf{y}) es:

$$\mathbf{y} = f(\mathbf{z}) = f(\mathbf{X}^T \mathbf{w}) \quad (3.9)$$

Con $f : \mathbb{R} \rightarrow \mathbb{R}$ la función de activación no lineal de las neuronas de la capa de salida.



(a) Arquitectura de la red neuronal de una sola capa [14]



(b) Estimación del error en la salida antes de la función de activación no-lineal [13]

El error en la salida después de la función de activación se describe como:

$$\mathcal{E} = \mathbf{d} - \mathbf{y} = \mathbf{d} - f(\mathbf{z}) = \mathbf{d} - f(\mathbf{X}^T \mathbf{w}) \quad ^1 \quad \mathcal{E} \in \mathbb{R}^{n \times 1} \quad (3.10)$$

De modo que el error cuadrático medio MSE queda definido por:

$$\text{MSE} = \sum_{i=1}^n \mathcal{E}_i^2 = \|\mathcal{E}\|^2 \quad (3.11)$$

Podemos plantear ahora el error de salida **antes** de pasar por la función de activación como:

$$\bar{\mathcal{E}} = \bar{\mathbf{d}} - \mathbf{z} = f^{-1}(\mathbf{d}) - \mathbf{X}^T \mathbf{w} \quad \bar{\mathcal{E}} \in \mathbb{R}^{n \times 1} \quad (3.12)$$

¹Función de activación aplica componente a componente. $\mathbf{v} \in \mathbb{R}^n \rightarrow f(\mathbf{v}) = (f(v_1), f(v_2), \dots, f(v_n))^T$

Donde $f^{-1} : \mathbb{R} \rightarrow \mathbb{R}$ es la inversa de la función de activación. De (3.12) también obtenemos la siguiente relación:

$$z = \bar{d} - \bar{\mathcal{E}} \quad (3.13)$$

Lo que nos permite introducir la siguiente aproximación de primer orden en serie de Taylor para cada componente del vector de salida \mathbf{y} :

$$\mathbf{y} = f(\mathbf{z}) = f(\bar{\mathbf{d}} - \bar{\mathcal{E}}) \approx \underbrace{f(\bar{\mathbf{d}}) - f'(\bar{\mathbf{d}}) \circ \bar{\mathcal{E}}}_{\text{aprox. Taylor}} = f(\bar{\mathbf{d}}) - f'(\bar{\mathbf{d}}) \circ (\bar{\mathbf{d}} - \mathbf{z}) \quad (3.14)$$

Con $f' : \mathbb{R} \rightarrow \mathbb{R}$ la derivada de la función de activación y $A \circ B$ el producto de Hadamard o producto componente a componente.

Esta última ecuación nos permite, finalmente, obtener una expresión del error en la salida en función de las salidas reales y las predichas antes de pasar por las funciones de activación no lineales (lo que denominamos \mathbf{z}) del siguiente modo, sustituyendo el resultado anterior en la ecuación (3.10):

$$\begin{aligned} \mathcal{E} &= \mathbf{d} - \mathbf{y} = \mathbf{d} - f(\bar{\mathbf{d}}) + f'(\bar{\mathbf{d}}) \circ (\bar{\mathbf{d}} - \mathbf{z}) \\ &= \mathbf{d} - (f(f^{-1}(\bar{\mathbf{d}}))) + f'(\bar{\mathbf{d}}) \circ (\bar{\mathbf{d}} - \mathbf{z}) \\ &= \mathbf{d} - \mathbf{d} + f'(\bar{\mathbf{d}}) \circ (\bar{\mathbf{d}} - \mathbf{z}) \\ &= f'(\bar{\mathbf{d}}) \circ (\bar{\mathbf{d}} - \mathbf{z}) = f'(\bar{\mathbf{d}}) \circ (\bar{\mathbf{d}} - \mathbf{X}^T \mathbf{w}) \end{aligned} \quad (3.15)$$

Definiendo ahora una matriz diagonal \mathbf{F} formada por la derivada de la función de activación f sobre cada componente del vector $\bar{\mathbf{d}}$, es decir, $\mathbf{F} = \text{diag}(f'(\bar{\mathbf{d}}_1), f'(\bar{\mathbf{d}}_2), \dots, f'(\bar{\mathbf{d}}_n))$, podemos reescribir la ecuación anterior y el MSE visto en (3.11) como:

$$\mathcal{E} = \mathbf{F}(\bar{\mathbf{d}} - \mathbf{X}^T \mathbf{w})$$

$$\text{MSE} = \|\mathcal{E}\|^2 = \|\mathbf{F}(\bar{\mathbf{d}} - \mathbf{X}^T \mathbf{w})\|^2 = (\mathbf{F}(\bar{\mathbf{d}} - \mathbf{X}^T \mathbf{w}))^T (\mathbf{F}(\bar{\mathbf{d}} - \mathbf{X}^T \mathbf{w})) \quad (3.16)$$

Con la ecuación (3.16) podemos introducir ahora la **función objetivo** de este modelo, $\mathbf{J}(\mathbf{w})$, a la que se le añade un término de regularización para evitar el sobreajuste o *overfitting* basado en la normal L2 o *ridge*, $\mathbf{R} = \frac{1}{2} \mathbf{w}^T \mathbf{w}$, de modo que queda:

$$\mathbf{J}(\mathbf{w}) = \frac{1}{2} \left[(\mathbf{F}(\bar{\mathbf{d}} - \mathbf{X}^T \mathbf{w}))^T (\mathbf{F}(\bar{\mathbf{d}} - \mathbf{X}^T \mathbf{w})) + \lambda \mathbf{w}^T \mathbf{w} \right] \quad (3.17)$$

La función objetivo de la ecuación anterior tiene la ventaja de que, aún usando funciones de activación no lineales (f), es **convexa** y presenta un mínimo global que puede ser obtenido directamente mediante solución cerrada para un \mathbf{X} y \mathbf{d} dados.

Este óptimo global se puede alcanzar fácilmente tras igualar a cero la derivada parcial de \mathbf{J} respecto a los pesos. La derivación de dicha expresión resulta de aplicar la propiedad de linealidad, trasposición de producto y suma de matrices y la regla del producto. A continuación detallamos los pasos esenciales:

$$\mathbf{J}(\mathbf{w}) = \frac{1}{2} \left[(\bar{\mathbf{d}}^T \mathbf{F}^T - \mathbf{w}^T \mathbf{X} \mathbf{F})(\mathbf{F} \bar{\mathbf{d}} - \mathbf{F} \mathbf{X}^T \mathbf{w}) + \lambda \mathbf{w}^T \mathbf{w} \right]$$

$$\begin{aligned}
 \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} &= \frac{1}{2} \left[-\mathbf{X}\mathbf{F}(\mathbf{F}\bar{\mathbf{d}} - \mathbf{F}\mathbf{X}^T\mathbf{w}) + (\bar{\mathbf{d}}^T\mathbf{F}^T - \mathbf{w}^T\mathbf{X}\mathbf{F})(-\mathbf{F}\mathbf{X}^T) + 2\lambda\mathbf{w} \right] = 0 \\
 &= \frac{1}{2} \left[-\mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}} + \mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T\mathbf{w} - \underbrace{\bar{\mathbf{d}}^T\mathbf{F}^T\mathbf{F}\mathbf{X}^T}_{\mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}}} + \underbrace{\mathbf{w}^T\mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T}_{\mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T\mathbf{w}} + 2\lambda\mathbf{w} \right] = 0 \\
 &= \frac{1}{2} \left[-2\mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}} + 2\mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T\mathbf{w} + 2\lambda\mathbf{w} \right] = 0 \\
 &= -\mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}} + \mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T\mathbf{w} + \lambda\mathbf{w} = 0 \\
 &= -\mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}} + (\mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T + \lambda\mathbf{I})\mathbf{w} = 0
 \end{aligned}$$

Que permite la construcción del sistema de ecuaciones lineales equivalente, $A\mathbf{w} = b$, cuya solución resulta en los **pesos óptimos** del modelo, que minimizan el MSE visto anteriormente:

$$(\mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T + \lambda\mathbf{I})\mathbf{w} = \mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}} \quad (3.18)$$

El tamaño del sistema de ecuaciones (3.18) depende de m , el número de atributos del conjunto de datos, con una complejidad computacional superior de $O(m^3)$ para el cálculo de la inversa de la matriz $(\mathbf{X}\mathbf{F}\mathbf{F}\mathbf{X}^T + \lambda\mathbf{I})$. Esto implica gran eficiencia sobre conjuntos de datos con un alto número de muestras y pocos atributos pero pobre rendimiento sobre conjuntos con muchas características, como es el caso de los *microarrays*. Para obtener los pesos óptimos del modo más eficiente posible – independientemente del aspecto del conjunto de datos – se introdujo, en trabajos posteriores, una transformación del sistema lineal (3.18) a través de la factorización de la matriz $\mathbf{X}\mathbf{F}$, realizada mediante descomposición en valores singulares **SVD**:

$$\mathbf{X}\mathbf{F} = \mathbf{U}\mathbf{S}\mathbf{V}^T \quad \mathbf{U} \in \mathbb{R}^{m \times m} \quad \mathbf{V} \in \mathbb{R}^{n \times n} \quad \mathbf{S} \in \mathbb{R}^{m \times n}$$

Con \mathbf{U} y \mathbf{V} matrices ortogonales y \mathbf{S} una matriz diagonal, de modo que resulta en un nuevo sistema de ecuaciones lineal:

$$(\mathbf{U}\mathbf{S}\mathbf{V}^T\mathbf{F}\mathbf{X}^T + \lambda\mathbf{I})\mathbf{w} = \mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}} \quad (3.19)$$

Cuya solución:

$$\mathbf{w} = \mathbf{U}(\mathbf{S}\mathbf{S}^T + \lambda\mathbf{I})^{-1}\mathbf{U}^T\mathbf{X}\mathbf{F}\mathbf{F}\bar{\mathbf{d}} \quad (3.20)$$

Presenta ahora la ventaja de que, como \mathbf{S} es una matriz diagonal con $r \leq \min(m, n)$ elementos no nulos, las dimensiones efectivas de \mathbf{U} y \mathbf{S} son $\mathbb{R}^{m \times r}$ y $\mathbb{R}^{r \times r}$, respectivamente, lo que permite usar métodos SVD simplificados (sin calcular si quiera \mathbf{V}), cuya **complejidad computacional** se reduce a $O(mnr)$. Esta descomposición es la operación más costosa, ya que la inversa de la matriz $(\mathbf{S}\mathbf{S}^T + \lambda\mathbf{I})$, al ser diagonal para cualquier $\lambda \neq 0$, es trivial, al ser la inversa de las componentes diagonales, con complejidad $O(r)$.

Aprendizaje federado y cifrado

La solución anterior está formulada para el caso de un escenario centralizado, teniendo íntegramente a disposición la matriz de entrada del conjunto de datos \mathbf{X} . Para adaptarlo al escenario federado, particularmente al conocido como *Aprendizaje Federado Horizontal*, donde podemos suponer la matriz de entrada del *dataset* como particionada en P submatrices, de modo que $\mathbf{X} = [\mathbf{X}_1 | \mathbf{X}_2 | \dots | \mathbf{X}_P]$, con cada cliente albergando una de estas submatrices, es necesario adaptar el algoritmo del siguiente modo:

- Respecto a la matriz $\mathbf{X}\mathbf{F}$, factorizada mediante la descomposición en valores singulares SVD, se ha probado, como se estudia en el artículo publicado por M.A. Iwen et al [16], que dicha descomposición se puede obtener de un modo **incremental** y **distribuido**:

$$\begin{aligned} \text{SVD}(\mathbf{X}\mathbf{F}) &= \text{SVD}([\mathbf{X}_1\mathbf{F}_1 | \mathbf{X}_2\mathbf{F}_2 | \dots | \mathbf{X}_P\mathbf{F}_P]) \\ &= \text{SVD}([\mathbf{U}_1\mathbf{S}_1 | \mathbf{U}_2\mathbf{S}_2 | \dots | \mathbf{U}_P\mathbf{S}_P]) \end{aligned} \quad (3.21)$$

Es decir, dada una matriz dividida en P particiones, su descomposición SVD tiene los mismos valores singulares \mathbf{S} y valores singulares izquierdos \mathbf{U} que la matriz resultante de concatenar la descomposición del producto $\mathbf{U}_p\mathbf{S}_p$ de las submatrices.

- El término $\mathbf{X}\mathbf{FF}\bar{\mathbf{d}}$, que tanto en este trabajo como en el original, por simplicidad, denominamos $\mathbf{m} \in \mathbb{R}^{m \times 1}$, se calcula de manera distribuida mediante el sumatorio de un producto de matrices que solo contienen la información de las submatrices de cada cliente como factores, pudiendo, por tanto, computarse incrementalmente mediante la adición de la información local \mathbf{m}_p provista por cada nodo:

$$\mathbf{m} = \mathbf{X}\mathbf{FF}\bar{\mathbf{d}} = \sum_{p=1}^P \mathbf{X}_p\mathbf{F}_p\mathbf{F}_p\bar{\mathbf{d}}_p = \sum_{p=1}^P m_p \quad (3.22)$$

De esta manera, podemos crear un entorno federado con arquitectura cliente-servidor, en el que P participantes (nodos) – partiendo con la misma estructura de datos (en cuanto refiere a número de características) – entrenen colaborativamente el modelo de ML. Para hacerlo, cada cliente, usando su partición local de la matriz de datos de entrada \mathbf{X}_p , tendría que:

1. Calcular las submatrices \mathbf{U}_p , \mathbf{S}_p resultado de la descomposición SVD de $\mathbf{X}_p\mathbf{F}_p$.
2. Calcular el vector $\mathbf{m}_p = \mathbf{X}_p\mathbf{F}_p\mathbf{F}_p\bar{\mathbf{d}}_p$.

El coordinador/servidor, por su parte, recibiría las matrices y vectores \mathbf{U}_p , \mathbf{S}_p y \mathbf{m}_p de cada cliente y agregaría la información al modelo global usando las ecuaciones (3.21) y (3.22). Con la información agregada, ya podría calcular los **pesos óptimos** según lo visto en la ecuación (3.20).

El único aspecto restante es el **cifrado**. Aunque en el escenario federado trazado más arriba no hay transmisión alguna de datos en bruto a través de los nodos del modelo (garantizando la *privacidad por diseño*), se decide añadir una capa más de seguridad mediante el uso de cifrado homomórfico, optando por cifrar el vector \mathbf{m}_p de cada cliente por un par de motivos:

- El vector \mathbf{m}_p puede ser más vulnerable a posibles ataques que las matrices fruto de la descomposición; \mathbf{U}_p , \mathbf{S}_p ; que, sin la \mathbf{V}_p^T correspondiente, no valen para recuperar la factorización original.
- La sucesiva adición de vectores encriptados homomórficamente está completamente soportada e ilimitada.

Para representar vectores y matrices encriptados, así como las operaciones entre ellos, usaremos la siguiente notación, $[[\cdot]]$. El tipo de encriptación homomórfica usada es de tipo FHE, mediante el esquema propuesto por Cheon, Kim, Kim y Song (**CKKS**, siglas de los autores del estudio '*Homomorphic Encryption for Arithmetic of Approximate Numbers*' [6]), que permite la operación con números reales.

Algoritmos coordinador y clientes

A continuación presentamos los algoritmos para coordinador y cliente del modelo de partida **FedHEONN**:

Algoritmo 2 Pseudocódigo para cliente FedHEONN

```

1: Entrada para un cliente  $p$ :
2:    $\mathbf{X}_p \in \mathbb{R}^{m \times n_p}$                                {Matriz de datos locales con  $m$  atributos y  $n$  muestras}
3:    $\mathbf{d}_p \in \mathbb{R}^{n_p \times 1}$                          {Vector correspondiente con las salidas reales}
4:    $f$                                                        {Función de activación no lineal}
5: Salida:
6:    $\llbracket \mathbf{m}_p \rrbracket$                                 {Vector local  $\mathbf{m}$  encriptado}
7:    $\mathbf{U}\mathbf{S}_p$                                          {Matriz  $\mathbf{U}\mathbf{S}$  local}

function FEDHEONN_CLIENT( $\mathbf{X}_p, \mathbf{d}_p, f$ ) :
8:    $\mathbf{X}_p \leftarrow [\text{ones}(1, n_p); \mathbf{X}_p]$            {Se añade bias}
9:    $\bar{\mathbf{d}}_p \leftarrow f^{-1}(\mathbf{d}_p)$                   {Inversa de la función de activación}
10:   $\mathbf{f}_p \leftarrow f'(\bar{\mathbf{d}}_p)$                       {Derivada de la función de activación}
11:   $\mathbf{F}_p \leftarrow \text{diag}(\mathbf{f}_p)$                    {Matriz diagonal}
12:   $[\mathbf{U}_p, \mathbf{S}_p, \sim] \leftarrow \text{SVD}(\mathbf{X}_p \mathbf{F}_p)$  {SVD 'reducido'}
13:   $\mathbf{U}\mathbf{S}_p \leftarrow \mathbf{U}_p \text{diag}(\mathbf{S}_p)$        {Producto de matrices locales  $\mathbf{U}_p\mathbf{S}_p$ }
14:   $\mathbf{m}_p \leftarrow \mathbf{X}_p (\mathbf{f}_p \circ \mathbf{f}_p \circ \bar{\mathbf{d}}_p)$  {Vector local  $\mathbf{m}_p$ }
15:   $\llbracket \mathbf{m}_p \rrbracket \leftarrow \text{ckks\_encryption}(\mathbf{m}_p)$  {Encriptación CKKS del vector  $\mathbf{m}_p$ }
16: return  $\llbracket \mathbf{m}_p \rrbracket, \mathbf{U}\mathbf{S}_p$ 
end function

```

Podemos observar como, mientras cada cliente se limita a computar la matriz resultado del producto $\mathbf{U}_p\mathbf{S}_p$ y el vector encriptado $\llbracket \mathbf{m}_p \rrbracket$ a partir de sus datos locales; es el coordinador el que realiza la mayor parte del trabajo agregándolos para cada cliente y calculando la solución del sistema global, vista en la ecuación 3.20, una vez finalizado.

Sobre este último punto cabe una última observación; para el cálculo de los pesos óptimos (línea 19 en algoritmo 3) se producen **dos** multiplicaciones matriciales consecutivas, justo el límite superior de operaciones encadenadas de este tipo que se pueden realizar bajo el esquema CKKS de encriptación homomórfica que se usa en este trabajo, como se analizó en el artículo de Shereen Mohamed Fawaz et al [11]. Como dicha operación se realiza una vez agregada la información de los clientes, no depende del número de estos. La adición de vectores encriptados, como se menciona al final del apartado anterior, está ilimitada en el número máximo de operaciones sucesivas que se pueden realizar sin degenerar el resultado final.

3.6.2. Agregación parcial e incremental

La primera de las mejoras realizada en este trabajo sobre los algoritmos del modelo base **FedHEONN** para el cliente (algoritmo 2) y el coordinador (algoritmo 3) fue la implementación de la agregación parcial e incremental, aspecto que teóricamente encontramos dispuesto pero cuya realización no se había probado.

Algoritmo 3 Pseudocódigo para coordinador FedHEONN

```

1: Entrada:
2:    $\mathbf{M}_{list}$                                {Lista con los vectores  $\mathbf{m}$  encriptados de los distintos clientes}
3:    $\mathbf{U}\mathbf{S}_{list}$                          {Lista con las matrices  $\mathbf{US}$  de los distintos clientes}
4:    $\lambda$                                      {Hiperparámetro de regularización}

5: Salida:
6:    $[\mathbf{w}] \in \mathbb{R}^{m \times 1}$            {Pesos óptimos encriptados}

function FEDHEONN_COORDINATOR( $\mathbf{M}_{list}, \mathbf{U}\mathbf{S}_{list}, \lambda$ ) :
7: if existen matrices almacenadas  $[\mathbf{m}], \mathbf{US}$ : then
8:    $[\mathbf{m}] \leftarrow [\mathbf{m}]$  existente          {Vector  $\mathbf{m}$  inicializado}
9:    $\mathbf{US} \leftarrow \mathbf{US}$  existente          {Matriz  $\mathbf{US}$  inicializada}
10: else
11:    $[\mathbf{m}] \leftarrow \mathbf{0}$                   {Vector de ceros}
12:    $\mathbf{US} \leftarrow []$                          {Matriz vacía}
13: end if
14: for  $[\mathbf{m}_p], \mathbf{U}\mathbf{S}_p$  in  $(\mathbf{M}_{list}, \mathbf{U}\mathbf{S}_{list})$  do
15:    $[\mathbf{m}] \leftarrow [\mathbf{m}] + [\mathbf{m}_p]$           {Agregamos vectores  $\mathbf{m}_p$ }
16:    $[\mathbf{U}, \mathbf{S}, \sim] \leftarrow \text{SVD}([\mathbf{US} | \mathbf{U}_p \mathbf{S}_p])$  {SVD incremental}
17:    $\mathbf{US} \leftarrow \mathbf{U} \text{diag}(\mathbf{S})$           {Agregación de matriz  $\mathbf{US}$ }
18: end for
19:  $[\mathbf{w}] \leftarrow \mathbf{U} (\mathbf{SS}^T + \lambda I)^{-1} (\mathbf{U}^T [\mathbf{m}])$  {Calculamos pesos óptimos}
20: Guardamos  $[\mathbf{m}], \mathbf{US}$                       {Para agregar nueva información en el futuro}
21: return  $[\mathbf{w}]$ 
end function

```

Los cambios necesarios para dicho ajuste son mínimos: el algoritmo del cliente permanece invariable y, en el algoritmo del coordinador, desglosamos la agregación parcial – con la información proveniente de los clientes – del propio cálculo de los pesos óptimos del sistema del modelo global, como se aprecia en los algoritmos 4 y 5; siendo el único cambio significativo la separación de la matriz global existente \mathbf{US} en las matrices \mathbf{U} y \mathbf{S} , para poder acceder a ellas individualmente desde ambas funciones.

Esto permite, por un lado, **separar la agregación de la información de la obtención de los pesos óptimos**, que se puede realizar ahora en cualquier interludio de la progresiva incorporación de los datos de los clientes que, por otro lado, **ya no se tienen por que enviar en conjunto**; obligando a la espera y recopilación de las matrices y vectores $\mathbf{U}_p \mathbf{S}_p$ y \mathbf{m}_p de todos ellos (cuello de botella generado por el más lento de los clientes), si no que se pueden ir enviando lotes o pequeños grupos de ellos (incluso individualmente), según la disponibilidad presente o según finalicen sus respectivos entrenamientos locales.

Experimentalmente, tenemos la necesidad de corroborar que esta aproximación a la agregación incremental de la información contenida en las matrices locales \mathbf{U}_p , \mathbf{S}_p y \mathbf{m}_p por lotes sea igualmente efectiva que la vista en la ecuación (3.21). Para ello, supongamos ahora que los P nodos (respectivas particiones de \mathbf{X}) se agrupan en K grupos, de modo que $\mathbf{X} = [\mathbf{X}_1 | \dots | \mathbf{X}_P] = [\mathbf{X}^1 | \dots | \mathbf{X}^K]$, donde cada grupo \mathbf{X}^k puede albergar un subconjunto disjunto de nodos (es decir, distintas submatrices \mathbf{X}_p) del conjunto total, tal

que $\mathbf{X}^k = \cup \mathbf{X}_i \quad \forall i \in \mathcal{I}_k$, donde \mathcal{I}_k es el conjunto de índices de los nodos/particiones pertenecientes al grupo k . Evidentemente, siempre se cumple $K \leq P$, igualándose sólo en el caso de que cada grupo contenga un único nodo, aunque normalmente se tendrá que $K \ll P$. Es necesario demostrar que:

$$\begin{aligned} \text{SVD}(\mathbf{X}\mathbf{F}) &= \text{SVD}([\mathbf{X}_1\mathbf{F}_1 | \dots | \mathbf{X}_P\mathbf{F}_P]) = \text{SVD}([\mathbf{U}_1\mathbf{S}_1 | \dots | \mathbf{U}_P\mathbf{S}_P]) \\ &= \text{SVD}([\mathbf{X}^1\mathbf{F}^1 | \dots | \mathbf{X}^K\mathbf{F}^K]) = \text{SVD}([\mathbf{U}^1\mathbf{S}^1 | \dots | \mathbf{U}^K\mathbf{S}^K]) \end{aligned} \quad (3.23)$$

Donde $\mathbf{U}^k\mathbf{S}^k$ son, a su vez, las matrices de vectores singulares izquierdos y valores singulares correspondientes al $\text{SVD}(\cup [\mathbf{X}_i\mathbf{F}_i]) \quad \forall i \in \mathcal{I}_k$.

Algoritmo 4 Pseudocódigo para la **agregación parcial** en coordinador FedHEONN

```

function FEDHEONN_COORDINATOR:AGREGACION_PARCIAL( $\mathbf{M}_{grupo}, \mathbf{U}\mathbf{S}_{grupo}$ ) :
1: if existen matrices almacenadas  $\llbracket \mathbf{m} \rrbracket, \mathbf{U}, \mathbf{S}$ : then
2:    $\llbracket \mathbf{m} \rrbracket \leftarrow \llbracket \mathbf{m} \rrbracket$  existente                                {Vector  $\mathbf{m}$  inicializado}
3:    $\mathbf{U} \leftarrow \mathbf{U}$  existente                                {Matriz  $\mathbf{U}$  inicializada}
4:    $\mathbf{S} \leftarrow \mathbf{S}$  existente                                {Matriz  $\mathbf{S}$  inicializada}
5: else
6:    $\llbracket \mathbf{m} \rrbracket \leftarrow \mathbf{0}$                                 {Vector de ceros}
7:    $\mathbf{U} \leftarrow []$                                          {Matriz vacía}
8:    $\mathbf{S} \leftarrow []$ 
9: end if
10: for  $\llbracket \mathbf{m}_p \rrbracket, \mathbf{U}\mathbf{S}_p$  in  $(\mathbf{M}_{grupo}, \mathbf{U}\mathbf{S}_{grupo})$  do
11:    $[\mathbf{U}, \mathbf{S}, \sim] \leftarrow \text{SVD}([\mathbf{U}\mathbf{S} | \mathbf{U}\mathbf{S}_p])$           {Agregación incremental de matriz  $\mathbf{U}\mathbf{S}_p$ }
12:    $\llbracket \mathbf{m} \rrbracket \leftarrow \llbracket \mathbf{m} \rrbracket + \llbracket \mathbf{m}_p \rrbracket$           {Agregamos vectores  $\mathbf{m}_p$ }
13: end for
14: Guardamos  $\llbracket \mathbf{m} \rrbracket, \mathbf{U}, \mathbf{S}$                                 {Para agregar nueva información en el futuro}
end function

```

Algoritmo 5 Pseudocódigo para calcular **pesos óptimos** en coordinador FedHEONN

```

function FEDHEONN_COORDINATOR:CALCULAR_PESOS( $\lambda$ ) :
1: if existen matrices almacenadas  $\llbracket \mathbf{m} \rrbracket, \mathbf{U}, \mathbf{S}$ : then
2:    $\llbracket \mathbf{w} \rrbracket \leftarrow \mathbf{U} (\mathbf{S}\mathbf{S}^T + \lambda I)^{-1} (\mathbf{U}^T \llbracket \mathbf{m} \rrbracket)$           {Calculamos pesos óptimos}
3:   return  $\llbracket \mathbf{w} \rrbracket$                                          {Vector  $\mathbf{w}$  encriptado}
4: else
5:   raise error                                         {Lanzamos error}
6: end if
end function

```

3.6.3. Ensamblaje

El siguiente desarrollo juega un papel central en la investigación realizada en este trabajo. Se trata de incorporar una técnica de ensamblaje al algoritmo para comprobar si podemos mejorar la **capacidad de representación** de esta sencilla red neuronal lineal (se puede imaginar como

un *perceptrón* de una sola capa), y darle así la versatilidad necesaria para que destaque en escenarios federados compuestos por nodos de escasa potencia computacional que equilibren eficiencia y eficacia.

El ensamblaje o *ensemble* son métodos que, en esencia, combinan múltiples modelos (denominados *estimadores base*) para mejorar el rendimiento y exactitud de las predicciones. La idea principal es que, al combinar varios modelos, bien con algoritmos de aprendizaje distintos, bien con matrices de datos (*datasets*) diferentes, se pueden reducir los errores que podrían darse al usar un único modelo.

Aunque existen múltiples técnicas de ensamblaje, las dos fundamentales y más conocidas son las de tipo **bagging** y **boosting**. Esta última construye una serie de modelos secuencialmente donde cada uno trata de corregir los fallos cometidos por los anteriores, poniendo énfasis en las predicciones erróneas realizadas; mientras que, con la primera, se crean simultáneamente versiones de un modelo utilizando subconjuntos diferentes de los datos de entrenamiento, promediando sus predicciones.

El **boosting** tiene un enfoque, durante su entrenamiento, *secuencial*, ya que depende de las predicciones del estimador anterior para poder preparar y equilibrar el conjunto de datos de entrenamiento del siguiente, forzando un mayor número de rondas de comunicación entre servidor y clientes en un escenario federado; mientras que el **bagging**, al contrario, posibilita la *parallelización* de dicho entrenamiento, ya que los estimadores base son independientes unos de otros.

Como el modelo de partida presenta unas características muy singulares como la naturaleza no iterativa de su entrenamiento, debido a la solución cerrada de los pesos óptimos globales, nos pareció más apropiado implementar una técnica de tipo *bagging* a FedHEONN, que una de tipo *boosting*, aunque suelen presentar mejores rendimientos.

Random Patches: una generalización del Bagging

La técnica específica que se ha implementado en este trabajo es la denominada **Random Patches**, presentada en el artículo publicado por Gilles Louppe et al. [23], como la máxima generalización del *bootstrap aggregating*. Básicamente se crea un *ensemble* de estimadores base entrenados a partir de ciertos *parches* de los datos de entrenamiento, generados a través de la extracción aleatoria de subconjuntos de muestras y atributos. En dicho estudio se analiza como esta técnica iguala e incluso mejora métricas obtenidas por otros métodos de este tipo, al tiempo que reduce las necesidades de memoria.

Descripción

La técnica propuesta puede ser descrita formalmente del siguiente modo: sea $R(\rho_s, \rho_f, D)$ el conjunto de todos los *parches* de tamaño $\rho_s N_s \times \rho_f N_f$ que se pueden extraer del conjunto de datos D , donde N_s es el número de muestras y N_f el número de atributos en D y donde $\rho_s, \rho_f \in [0, 1]$ son los hiperparámetros que controlan, respectivamente, el número de muestras y atributos en un *parche*. Es decir, sea $R(\rho_s, \rho_f, D)$ la serie de todos los posibles subconjuntos de $\rho_s N_s$ muestras y $\rho_f N_f$ atributos de D , el método se aplica del siguiente modo:

1. **Extraer** un *parche* $r \sim U(R(\rho_s, \rho_p, D))$ aleatoriamente según una distribución uniforme del conjunto.
2. **Entrenar** un estimador base con el parche seleccionado r .

3. **Repetir** los pasos 1-2 para un número T de estimadores.
4. **Promediar** las predicciones de los T estimadores base, en caso de regresión, o usar un voto mayoritario para la clasificación.

Memoria

A parte de ser una estrategia competitiva a la hora de implementar un ensamblaje de tipo *bagging*, otra ventaja de esta técnica es que, tanto en el contexto de grandes conjuntos de datos, como en el de dispositivos embebidos bajo fuertes restricciones de memoria; es decir, en toda situación en la que el tamaño del conjunto de datos es mucho mayor que la memoria disponible, usar parches aleatorios del tamaño apropiado, sacados de los datos de entrenamiento, para construir estimadores base puede resultar en un modelo final tan bueno, o más, que como si todos los datos se hubieran podido cargar y usar directamente.

Formalmente supongamos que, cuantificando la memoria utilizada por una unidad de datos como M , nos encontramos en un contexto tal que estamos restringidos a un umbral superior determinado por M_{max} ; de modo que M_{max} se pueda entender como el número total de unidades de memoria que se pueden cargar y utilizar en un determinado modelo.²

En este escenario, la cantidad de memoria requerida por un *random-patch* sería $(\rho_s N_s)(\rho_f N_f)$ y, por tanto, restringir la memoria en M_{max} es equivalente a limitar el tamaño relativo del *parche* a:

$$\rho_s \rho_f \leq \frac{M_{max}}{N_s N_f}$$

Los **experimentos** realizados con esta técnica pretender comprobar:

- Que, efectivamente, con este método se pueden conseguir **mejores métricas y rendimientos** que bajo la versión original de FedHEONN.
- Analizar la **sensibilidad de distintos conjuntos** $(\rho_s, \rho_f, Acc_D(\rho_s, \rho_f) \mid \forall \rho_s, \rho_f)$ sobre varios *datasets* D , siendo $Acc_D(\rho_s, \rho_f)$ la exactitud media del ensamblaje construido con parches aleatorios de tamaño relativo $\rho_s \rho_f$.
- Comparar la **eficiencia** entre los mejores resultados obtenidos bajo esta técnica y los del modelo original, así como métricas equivalentes a estas últimas pero con mayor eficiencia y menor consumo de memoria (a base de usar un *ensemble* de pocos parches y pequeño tamaño relativo).

Implementación en FedHEONN

A continuación se describe el método de *ensemble* desarrollado en este proyecto empleando como modelo base el FedHEONN. En esencia, los cambios necesarios en los algoritmos de cliente y coordinador fueron los siguientes:

Cliente

- Añadimos el conjunto de hiperparámetros $(\rho_s, \beta_s, \rho_f, \beta_f, T)$ para precisar el tamaño relativo de los *parches* $[\rho_s, \rho_f]$, si estos se extraen **con o sin reemplazo**, con los *flags* $[\beta_s, \beta_f]$, y el número de estimadores base T involucrados en el ensamblaje.

²No tenemos en cuenta la memoria utilizada por el propio modelo durante el transcurso de las operaciones inherentes a su algoritmo de entrenamiento.

- Se *envuelve* la función descrita en el algoritmo 2 por una encargada de extraer aleatoriamente los *parches* del conjunto de datos local, pasarlo por la función anterior de entrenamiento (*fit*) y acumular sus resultados en una lista, para posterior envío al coordinador.
- Por último, y del mismo modo, se *superpone* a la función de predicción (*predict*), que introducimos en el algoritmo 7, otra encargada de obtener las predicciones para cada conjunto de pesos óptimos de cada estimador base T y, con los resultados, realizar el promedio o voto mayoritario dependiendo de si estamos ante un problema de regresión o de clasificación.

Coordinador

- El único hiperparámetro necesario en el coordinador es un *flag*, denominado *ensemble*, para indicar si estamos ante un caso de ensamblaje o no, ya que el número de estimadores base T se puede deducir a través de la longitud de la lista de matrices $\llbracket \mathbf{m}_p \rrbracket$, $\mathbf{U} \mathbf{S}_p$ entregada por cada cliente.
- Se aplica por separado la agregación parcial y cálculo de pesos óptimos en cada i -ésimo $\llbracket \mathbf{m}_p^i \rrbracket$, $\mathbf{U} \mathbf{S}_p^i | \forall i = 1 \dots T$ de cada cliente P , obteniendo una lista final de pesos óptimos $[\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_T]$.

En definitiva, se trata de generalizar las fases de entrenamiento, agregación y cálculo de pesos para realizarlas T veces con cada parche aleatorio r extraído del *dataset* D_p local correspondiente a cada cliente.

Sincronización de características

La única dificultad y sobrecoste que implica este desarrollo radica en la necesidad de coordinar, entre los distintos nodos o clientes, exactamente cuáles son los $\lfloor \rho_f N_f \rfloor$ atributos aleatorios seleccionados para la extracción del parche r - i -ésimo de cada estimador base T , ya que no podemos realizar la correcta agregación entre clientes de las matrices $\llbracket \mathbf{m}_p^i \rrbracket$, $\mathbf{U} \mathbf{S}_p^i$ si estas no son el resultado del entrenamiento local sobre un $\mathbf{X}_p^i \in \mathbb{R}^{(\rho_s m) \times (\rho_f n_p)}$ de las mismas características.

Para ello se delegó esta tarea al coordinador, que ofrece ahora una función auxiliar capaz de generar una lista de T series con los índices de los atributos seleccionados aleatoriamente, basándose en la fracción y número de atributos (ρ_f , N_f) entregado como argumento y de si se permite reemplazo o no (β_f).

Esta lista es luego **enviada a todos los clientes** implicados en el aprendizaje federado, suponiendo una ronda de comunicación previa al entrenamiento de la red.

Algoritmos

A continuación presentamos los algoritmos modificados para el cliente FedHEONN en sus tareas de entrenamiento y predicción, respectivamente algoritmos 6 y 7; para terminar con los correspondientes a la agregación parcial y cálculo de pesos óptimos, algoritmos 8 y 9, en el coordinador.

Algoritmo 6 Pseudocódigo para entrenamiento de cliente FedHEONN con **ensamblaje**

1: **Entrada** para un cliente p :
2: $\mathbf{X}_p \in \mathbb{R}^{m \times n_p}$, $\mathbf{d}_p \in \mathbb{R}^{n_p \times 1}$, f {Matriz de datos, vector de salidas y f.act.}
3: $T \in \mathbb{N}$ {Número de estimadores base}
4: $\rho_s \in (0, 1]$ {Proporción del tamaño de muestras del parche}
5: $\beta_s := \text{true or false}$ {Muestreo con reemplazo para registros}
6: $\rho_f \in (0, 1]$ {Proporción del tamaño de atributos del parche}
7: $\beta_f := \text{true or false}$ {Muestreo con reemplazo para atributos}

8: **Salida**:
9: $\{\mathbf{m}_p^i\}_{i=1}^T$, $\{\mathbf{U}\mathbf{S}_p^i\}_{i=1}^T$ {Lista de T vectores \mathbf{m}_p y matrices $\mathbf{U}\mathbf{S}_p$ }

10:

function FEDHEONN_CLIENT:FIT(\mathbf{X}_p , \mathbf{d}_p , f , T , ρ_s , β_s , ρ_f , β_f) :

11: $\{\mathcal{I}_f\}_{i=1}^T \leftarrow \text{FEDHEONN_COORD:random_attributes}(T, \rho_f, \beta_f, m)$ {Lista con T series de índices aleatorios de atributos, generados y compartidos por el coordinador}

12: **for** i in T **do**

13: $\mathcal{I}_f \leftarrow \{\mathcal{I}_f\}_i$ {Serie i -ésima de índices de atributos aleatorios}

14: $\mathcal{I}_s \leftarrow \text{random_choice(array}=\mathbf{X}_p, \text{size}=\rho_s \times n_p, \text{replace}=\beta_s)$ {Índices muestras}

15: $\mathbf{X}_p^i \leftarrow \mathbf{X}_p[\mathcal{I}_f, \mathcal{I}_s]$ {Indexamos *parche* de matriz de datos local}

16: $\mathbf{d}_p^i \leftarrow \mathbf{d}_p[\mathcal{I}_f, \mathcal{I}_s]$ {Indexamos *parche* del vector de salidas local}

17: $\{\mathbf{m}_p^i\}, \{\mathbf{U}\mathbf{S}_p^i\} \leftarrow \text{FEDHEONN_CLIENT}(\mathbf{X}_p^i, \mathbf{d}_p^i, f)$ {Algoritmo 2}

18: **end for**

19: **return** $\{\mathbf{m}_p^i\}_{i=1}^T$, $\{\mathbf{U}\mathbf{S}_p^i\}_{i=1}^T$ {Devolvemos lista de vectores y matrices}

end function

Algoritmo 7 Pseudocódigo para predicción en cliente FedHEONN con **ensamblaje**

```

1: Entrada para un cliente  $p$ :
2:    $\mathbf{X}_p^{test} \in \mathbb{R}^{m \times n_p}$                                 {Matriz de datos local de test o predicción}
3:    $[\mathbf{w}] \in \mathbb{R}^{m \times 1}$                                {Vector de pesos óptimos encriptado}
4:    $f$                                                        {Función de activación}
5:    $T \in \mathbb{N}$                                          {Número de estimadores base}
6:    $\{\mathcal{I}_f\}_{i=1}^T$           {Serie de índices aleat. de atributos usados durante el entrenamiento}
7: Salida:
8:    $\mathbf{y} \in \mathbb{R}^{n_p \times 1}$                                 {Vector de predicciones sobre test}
9:

function FEDHEONN_CLIENT:PREDICT_ENSEMBLE( $\mathbf{X}_p^{test}$ ,  $[\mathbf{w}]$ ,  $f$ ,  $T$ ,  $\{\mathcal{I}_f\}_{i=1}^T$ ) :
10:   $\mathbf{w} \leftarrow \text{ckks\_decryption}([\mathbf{w}])$            {Desencriptamos vector de pesos óptimos}
11:  for  $i$  in  $T$  do
12:     $\mathcal{I}_f \leftarrow \{\mathcal{I}_f\}_i$                       {Serie  $i$ -ésima de índices de atributos aleatorios}
13:     $\mathbf{X}_p \leftarrow \mathbf{X}_p^{test}[\mathcal{I}_f, :]$      {Indexamos características de la matriz de datos de predicción}
14:     $\{\mathbf{y}_i\} \leftarrow \text{FEDHEONN_CLIENT:PREDICT}(\mathbf{X}_p, \mathbf{w}, f)$  {Pred. estimador base  $i$ -ésimo}
15:  end for
16:  if CLASIFICADOR then {                                         Para un problema de clasificación}
17:     $\mathbf{y}^{test} \leftarrow \arg \max[\mathbf{y}_1 | \mathbf{y}_2 | \dots | \mathbf{y}_T]$  {Voto mayoritario}
18:  else if REGRESOR then {                                         Para un problema de regresión}
19:     $\mathbf{y}^{test} \leftarrow \frac{1}{T} \sum_{i=1}^T \mathbf{y}_i$                 {Promedio}
20:  end if
21:  return  $\mathbf{y}^{test}$                                               {Vector de predicciones}
end function
22:

function FEDHEONN_CLIENT:PREDICT( $\mathbf{X}_p$ ,  $\mathbf{w}$ ,  $f$ ) :
23:   $\mathbf{y} \leftarrow f(\mathbf{X}_p^T \mathbf{w})$                                 { $f$  aplica componente a componente}
24:  return  $\mathbf{y}$ 
end function

```

Algoritmo 8 Pseudocódigo de agregación parcial en coordinador FedHEONN con **ensamblaje**

```

1: Entrada:
2:    $\{\mathbf{M}_{grupo}\}$                                 {Lista3 de listas con lotes de vectores encriptados  $\{[\mathbf{m}_p^i]\}_{i=1}^T$ }
3:    $\{\mathbf{U}\mathbf{S}_{grupo}\}$                             {Lista de listas con lotes de matrices locales  $\{[\mathbf{U}\mathbf{S}_p^i]\}_{i=1}^T$ }
function FEDHEONN_CLIENT:AGG_PARCIAL_ENSEMBLE( $\{\mathbf{M}_{grupo}\}$ ,  $\{\mathbf{U}\mathbf{S}_{grupo}\}$ ) :
4:   $T \leftarrow \text{length}(\{\mathbf{M}_{grupo_1}\})$            {Dedujimos nº de estimadores base}
5:  for  $i$  in  $T$  do
6:     $\mathbf{M}_{grupo} \leftarrow \{\mathbf{M}_{grupo_i}\}$            {Seleccionamos lote  $i$ -ésimo}
7:     $\mathbf{U}\mathbf{S}_{grupo} \leftarrow \{\mathbf{U}\mathbf{S}_{grupo_i}\}$ 
8:     $\text{FEDHEON_COORDINATOR:AGREGACION_PARCIAL}(\mathbf{M}_{grupo}, \mathbf{U}\mathbf{S}_{grupo})$  {Algoritmo 4}
9:  end for

```

³Tanto $\{\mathbf{M}_{grupo}\}$ como $\{\mathbf{U}\mathbf{S}_{grupo}\}$ representan una lista de clientes, derivados de un determinado grupo k , y aportando cada uno, a su vez, un lote de matrices $\{[\mathbf{m}_p^i]\}_{i=1}^T$, $\{\mathbf{U}\mathbf{S}_p^i\}_{i=1}^T$ de sus respectivos estimadores base T .

Algoritmo 9 Pseudocódigo cálculo pesos óptimos en coord. FedHEONN con ensamblaje

```

1: Salida:
2:    $\{\llbracket w \rrbracket\}_{i=1}^T$            {Lista de vectores encriptados con los pesos óptimos}
function FEDHEONN_CLIENT:CALCULAR_PESOS_ENSEMBLE() :
3: if existen matrices almacenadas  $\{\llbracket m \rrbracket\}$ ,  $\{\mathbf{U}\}$ ,  $\{\mathbf{S}\}$ : then
4:    $T \leftarrow \text{length}(\{\llbracket m \rrbracket\})$            {Deducimos nº estimadores base}
5:   for  $i$  in  $T$  do
6:      $\llbracket m \rrbracket \leftarrow \{\llbracket m \rrbracket_i\}$   {Obtenemos matrices y vectores  $i$ -ésimos globales almacenados}
7:      $\llbracket \mathbf{U} \rrbracket \leftarrow \{\llbracket \mathbf{U} \rrbracket_i\}$ 
8:      $\llbracket \mathbf{S} \rrbracket \leftarrow \{\llbracket \mathbf{S} \rrbracket_i\}$ 
9:      $\llbracket w_i \rrbracket \leftarrow \text{FEDHEON_COORDINATOR:CALCULAR_PESOS}()$            {Algoritmo 5}
10:    end for
11:    return  $\{\llbracket w_1 \rrbracket, \llbracket w_2 \rrbracket, \dots, \llbracket w_T \rrbracket\}$ 
12: else
13:   raise error                                {Lanzamos error}
14: end if

```

3.6.4. Paralelismo

La siguiente mejora realizada al modelo FedHEONN con ensamblaje fue la capacidad de parallelizar el entrenamiento, en el cliente, y la agregación parcial y calculo de pesos óptimos, en el coordinador.

Debido al incremento en el tiempo global de computo del entorno federado para casos con un gran número de estimadores base T , decidimos implementar esta solución como un modo de reducirlo y volverlo más eficiente, aprovechando la independencia entre estimadores que ofrece la técnica de *bagging* aplicada anteriormente (cada estimador base i -ésimo del nodo p puede entrenar/agregar/calcularse partiendo únicamente de su conjunto de datos \mathbf{X}_p^i local).

La única precaución durante su implementación fue asegurarse de mantener el orden de partida de los datos devueltos por los distintos procesos paralelos iniciados (*pool de procesos*), cuyos tiempos y orden de finalización son imprevisibles *a priori*, de modo que no mezclemos el orden de los estimadores base de ningún cliente.

Con esta mejora se espera aprovechar la capacidad de los dispositivos físicos con procesadores multi-núcleo (*multi-core*), que, aunque hace años solo estaban presentes en equipos personales, cada vez más se encuentran en todo tipo de sistemas móviles y embebidos, como, por ejemplo, las *Raspberry Pi* y *Jetson Nano* descritas en la sección 3.4.2.

Experimentalmente pretendemos comprobar que, a través de la paralelización:

- Podemos **acelerar**, individualmente y en conjunto, los procesos de entrenamiento, agregación y calculo de pesos óptimos.
- Comprobar a partir de que número de estimadores base T , para cierto par de $[\rho_s, \rho_f]$ y dataset D , **compensa** la paralelización (debido a la *sobrecarga* producida por la creación y manejo del *pool* de procesos).
- Obtener así una especie de *regla general* a partir de la cual es aconsejable activar dicha paralelización.

Implementación en FedHEONN

Los totalidad de los detalles técnicos del desarrollo de esta mejora escapan a la envergadura de esta sección, donde solo pretendemos repasar, a modo de memoria, los procedimientos utilizados, las trabas encontradas y las soluciones aplicadas para sortearlas.

La paralelización se realizó a través de la librería estándar de Python `multiprocessing`, que presenta una API de alto nivel para el inicio y manejo de procesos. La idea original consistía en crear un *pool* de procesos, tantos como *cores* físicos dispone el CPU, para que se fuesen encargando de entrenar, agregar y calcular los pesos óptimos de cada estimador base involucrado en el *ensemble* del modelo FedHEONN.

El principal problema que se nos presentó con esta aproximación fue que la librería de encriptación homomórfica con la que estuvimos trabajando, TenSEAL, **no soporta** que las clases y objetos que define fuesen *serializados*⁴ a través del módulo estándar que usa Python para este tipo de protocolos, `pickle`.

Como los distintos procesos creados por el módulo de `multiprocessing` **no comparten** el mismo espacio de memoria, cuando intentamos paralelizar el método de un objeto, Python intenta *serializar* esta instancia, además de sus argumentos, usando `pickle`, para compartirlos con los otros procesos creados; originando el problema descrito anteriormente.

La librería **TenSEAL** expone sus propios métodos de serialización y de-serialización (*reconstrucción*) de objetos de tipo *CKKS* (vectores encriptados de números reales) y del propio *contexto* de TenSEAL (custodia las llaves pública y privada de dicho esquema), por lo que, para implementar la paralelización en cliente y coordinador, hubo que realizar estas operaciones de manera programática mediante dichos métodos; de igual modo que lo hubiera hecho Python automáticamente a través de `pickle`. Esto colleva serializar fuera los argumentos que fuesen vectores encriptados, además del contexto de TenSEAL asociado, y reconstruirlos una vez dentro de toda función susceptible de ser paralelizable.

Esto implicó un breve estudio secundario, no demasiado exhaustivo pero concluyente, para determinar que tipo de operaciones eran más costosas y saber así cuales merecían la paralelización, debido al añadido que suponen las operaciones de serialización y reconstrucción de los vectores involucrados.

La encriptación de vectores y su adición son operaciones sencillas, mientras que encontramos la reconstrucción y serialización, tanto de contexto como vector, como operaciones muy costosas, al igual que la multiplicación de vectores encriptados. Con ello en mente, los cambios introducidos se resumen en:

Cliente

- La encriptación de los distintos vectores $\llbracket \mathbf{m}_p^i \rrbracket$, al final del **entrenamiento** del cliente, se realiza ajena a cualquier parte de la paralelización, ya que el pequeño coste de dicha operación no compensa la *sobrecarga* añadida por la serialización y reconstrucción del contexto necesario para ello.
- De esta manera, la paralelización en el cliente se realiza para toda la función descrita en

⁴Se denomina **serialización** al proceso encargado de convertir un objeto o estructura, en un determinado lenguaje de programación, a un formato que permita su almacenamiento, transmisión y futura reconstrucción a su estado original. Es muy útil para guardar el *estado* de un objeto o enviarlo a través de la red.

el algoritmo 2, exceptuando la línea 15.

Coordinador

- De manera muy parecida, en la **agregación parcial** se paraleliza todo el proceso involucrado con las matrices globales US^i y las distintas US_p^i de cada cliente (SVD), dejando la suma del término global $\llbracket m^i \rrbracket$ con los diferentes $\llbracket m_p^i \rrbracket$ aparte, ya que la suma de vectores encriptados CKKS es una operación sencilla, y no compensa su serialización. Equivale a paralelizar solo las líneas 12-14 del algoritmo 4.
- Para el **cálculo de pesos óptimos**, sin embargo, siempre que el número de estimadores base T sea mayor al número de *cores* físicos disponibles $\frac{CPU}{cores}$, paralelizamos toda la función, agrupándose los estimadores en lotes equitativos a repartir entre tantos procesos como *cores*, minimizando así el *sobrecoste* de tener que serializar y reconstruir el contexto en cada proceso iniciado, ya que las dos multiplicaciones de matrices con vector encriptado involucradas en el cálculo de cada w_i (línea 2 en el algoritmo 5) compensa el coste computacional.

3.6.5. Prototipo desarrollado para el entorno federado

Por último, desarrollamos una aplicación prototipo servidor-cliente, para demostrar el uso del modelo bajo un escenario realista de entorno federado, ya que todos los trabajos previos hasta la fecha habían simulado, bajo un mismo equipo o dispositivo, tanto el coordinador como los distintos nodos involucrados en sus pruebas.

Nos decantamos por utilizar la librería **FastAPI** para el desarrollo rápido de una interfaz de programación (Application Programming Interface, API) de tipo *REST* (Representational State Transfer)⁵, que nos permita demostrar la agregación parcial de la información de los distintos clientes conectados a la red y la predicción de los pesos óptimos calculados con ella.

Uno de los problemas que se manifestó durante este desarrollo fue el de la **serialización**, mencionado anteriormente, ya que es a través de este como podemos enviar y recibir los vectores encriptados de los distintos clientes al servidor; por lo que nuestra aplicación tiene que gestionar, a mayores de las funciones propias del modelo FedHEONN, otras para el manejo, serialización y reconstrucción de dichos vectores.

Experimentalmente, el objetivo fundamental de esta aplicación es el de servir de **demos-trador** de un caso realista de entrenamiento y validación en red de un modelo federado, sin mayor ambición que la de su propio funcionamiento.

Descripción de la API

A continuación presentamos una breve lista y descripción de los métodos ofrecidos por la API del servidor y las funciones equivalentes añadidas al cliente para su consumo:

- **Servicios de disponibilidad y estado:**

⁵Una API de tipo REST es una interfaz que permite la comunicación entre sistemas utilizando HTTP, a través de peticiones estándar donde los recursos se representan en formatos como JSON o XML. Es un estilo de arquitectura que favorece la escalabilidad y simplicidad entre cliente y servidor.

GET	/ping	Ping	▼
GET	/status	Status	▼

Figura 3.31: Endpoints para *ping* y *status* del servidor

El primero es un servicio de *ping*, para saber si el servidor está levantado y activo, mientras que el segundo, *status*, devuelve un informe del estado actual del servidor, con, entre otros, el contexto TenSEAL seleccionado, el número de clientes agregados y procesados, etc., como veremos en la demostración del siguiente capítulo.

- **Servicios de selección y descarga de datasets D_p :**

PUT	/dataset/{dataset_name}	Select Dataset	▼
GET	/dataset/load	Load Dataset	▼
GET	/dataset/fetch	Fetch Dataset	▼
GET	/dataset/fetch/test	Fetch Dataset Test	▼

Figura 3.32: Endpoints para la selección, carga y reparto de los datos de test y entrenamiento

Estos servicios son los únicos que no tendrían cabida en un escenario realista, ya que lo que ofrecemos con ellos es la posibilidad de cargar distintos conjuntos de datos en el propio servidor (específicamente los vistos en la tabla 3.8) y poder descargar fragmentos de distinto tamaño en cada cliente.

Esto se consigue mediante sucesivas llamadas a un par de *endpoints* (/dataset/fetch y dataset/fetch/test) hasta que se agoten los datos. Estos servicios se implementaron para poder realizar experimentos con los clientes bajo distintos tamaños de datos de entrenamiento en cada uno de ellos.

- **Servicios de gestión de contexto:**

POST	/context Upload Context	▼
GET	/context/{ctx_name} Get Context	▼
PUT	/context/{ctx_name} Select Context	▼
DELETE	/context/{ctx_name} Erase Context	▼

Figura 3.33: Endpoints para la subida, selección, descarga y eliminación de contextos TenSEAL

Los servicios anteriores permiten la gestión de los contextos de TenSEAL en el servidor, necesarios para la serialización y reconstrucción de la información (vectores CKKS) enviada entre nodos y servidor.

La idea aquí sería que un cliente o entidad de confianza (*trusted party*) genere el contexto TenSEAL a utilizar (durante entrenamientos, agregación y cálculo de pesos óptimo); y que lo subiese al servidor (públicamente, sin la clave privada asociada) mediante el endpoint /context. Una vez seleccionado este en el servidor, el resto de clientes involucrados podrían descargarlo a través de /context/{ctx_name} y todos podrían realizar así las operaciones criptográficas necesarias.

- **Servicios de ajuste de hiperparámetros del coordinador:**

DELETE	/server/clean Clean Server	▼
PUT	/coordinator/parameters Set Coordinator Parameters	▼
GET	/coordinator/index_features Send Index Features	▼
POST	/coordinator/index_features Calculate Index Features	▼
GET	/coordinator/send_weights Send Weights	▼

Figura 3.34: Endpoints para el ajuste de hiperparámetros, sincronización de características y envío de pesos óptimos del coordinador

Los servicios relacionados con el coordinador FedHEONN permiten la actualización en el servidor, al cual está acoplado, de cualquiera de sus hiperparámetros (*función de activación, regularización, ensamblaje, etc.*), como detallamos en la figura 3.35a.

Además, se presenta también una fachada a las funciones de utilidad para la sincronización de características (vía /coordinator/index_features), necesaria en el caso de realizar el entrenamiento con ensamblaje (como vimos en la sección 3.6.3), que detallamos a continuación:

1. Un cliente o entidad de confianza pide al servidor que cree T listas con $\rho_f N_f$ índices de atributos aleatorios, con/sin reemplazo.
2. El resto de clientes descargan dichas listas para entrenar a sus estimadores base con \mathbf{X}_p^i semejantes.

PUT `/coordinator/parameters` Set Coordinator Parameters

Parameters

No parameters

Request body required `application/json`

Example Value | Schema

```
{
  "f": "string",
  "lam": 0,
  "encrypted": true,
  "sparse": true,
  "bagging": true,
  "parallel": true,
  "ctx_str": "string"
}
```

(a) Detalle del servicio de actualización de hiperparámetros

POST `/coordinator/index_features` Calculate Index Features

Parameters

No parameters

Request body required `application/json`

Example Value | Schema

```
{
  "n_estimators": 0,
  "n_features": 0,
  "p_features": 0,
  "b_features": true
}
```

(b) Detalle del servicio de sincronización de características

- **Servicios de agregación y cálculo de pesos óptimos:**

POST `/aggregate/partial` Aggregate Partial

GET `/aggregate/status` Check Status

Figura 3.36: Endpoints para la *agregación parcial* y comprobación de la cola de procesado

Estos servicios son esenciales en la obtención del **modelo global** federado. Al primero, `/aggregate/partial`, los diferentes clientes que constituyen la red federada deben enviar sus matrices $\llbracket \mathbf{m}_p^i \rrbracket$, $\mathbf{U} \mathbf{S}_p^i$; quedando encoladas para su procesamiento por el coordinador, y obteniendo una serie de identificadores *UUID* con los que pueden comprobar el estado de estas (pendientes, procesándose o ya procesadas).

El *endpoint* para la agregación parcial se ha diseñado de tal manera se encuentre **disponible** ante llamadas **concurrentes** y **consecutivas** (al igual que el resto de servicios del servidor), creando, en otro hilo (*thread*), un bucle acoplado a una cola (*queue*) que se encarga, de fondo, de ir agregando, incrementalmente, las matrices encoladas.

Cuando esta cola se encuentra **vacía**, es decir, estamos ante un interludio donde ya se ha procesado toda la información recibida hasta el momento, se lanza automáticamente el **proceso de cálculo de pesos óptimos**. Estos pesos óptimos se pueden descargar, mediante el servicio `/coordinator/send_weights`, visto en la figura 3.34, por los diferentes clientes, para realizar las predicciones oportunas con el modelo global.

En el caso de que los clientes tengan acceso a la **clave privada** del contexto de TenSEAL utilizado durante el entrenamiento y agregación, podrán realizar la predicción desencriptando los pesos óptimos recibidos del servidor, $\llbracket \mathbf{w} \rrbracket$, y luego aplicando $\mathbf{y} = f(\mathbf{z}) = f(\mathbf{X}^T \mathbf{w})$. En caso contrario tendrían que recurrir a la entidad de confianza (*trusted party*), depositaria de la clave privada, a la que podrían enviar, para mayor seguridad, el vector resultado del producto matricial de los pesos por la matriz de datos a predecir, $\llbracket \mathbf{z} \rrbracket$, recibir dicho vector desencriptado \mathbf{z} y, finalmente, aplicarle la función de activación $\mathbf{y} = f(\mathbf{z})$.

Para los clientes se creó una clase con los respectivos métodos necesarios para consumir y comunicarse fácilmente con el servidor, conocida la dirección *IP* y el puerto en el que este se encuentra escuchando, como se aprecia en la figura 3.37 (no mostramos todas las funciones).

```

16  class Client:
17      """Client to communicate with server for aggregation and evaluation"""
18      # Abel
19      def __init__(self, hostname: str, port: int): ...
20
21      # 4 usages # Abel
22      def ping(self) -> bool: ...
23
24      # 9 usages # Abel
25      def get_status(self) -> ServerStatus: ...
26
27
28
29
30
31
32
33
34
35
36

```

Figura 3.37: Clase cliente para consumo de API, con inicialización, *ping* y *status*

3.7. Resultados finales

En este capítulo presentamos los experimentos propuestos en los diferentes apartados de la sección anterior, manteniendo una estructura análoga y analizando los resultados obtenidos con ellos, dando fin así a la parte práctica de este trabajo y preparándonos para las conclusiones finales.

En cada sección detallaremos las pruebas realizadas y su implementación, así como los resultados esperados y obtenidos, finalizando con el nombre de los ficheros que albergan el código para su reproducción (bajo el directorio `examples` en la estructura listada en la sección 4.2).

3.7.1. Agregación parcial e incremental

Las pruebas realizadas para verificar el correcto funcionamiento de la agregación parcial constan de tres fases:

1. Para un cierto *dataset* D , realizamos lo que denominamos `global_fit`, donde simplemente entrenamos y predecimos, usando la versión original del modelo vista en el algoritmo 3, para un número dado de clientes P .
2. Seguidamente, realizamos un `incremental_fit`, agrupando los P clientes en lotes de tamaño N_g , y efectuando con ellos la agregación parcial, cálculo de pesos óptimos (algoritmos 4 y 5) y predicción sobre el mismo conjunto de *test*.
3. Por último, para un número distinto de clientes, P' , repetimos el proceso agrupándolos en lotes de tamaño aleatorio, que también entrenamos y testeamos incrementalmente del mismo modo.

Con dichas pruebas se pretende comparar que las métricas obtenidas bajo dos escenarios distintos son equivalentes independientemente del procedimiento utilizado. La **agregación global** con P clientes debería ser equivalente a la **incremental** de los mismos realizada en lotes de tamaño N_g , y esta última similar a la realizada con un número distinto de clientes P' agrupados aleatoriamente.

Estas pruebas se realizaron sobre 3 conjuntos de datos distintos, definidos en la tabla 3.8, y se encuentran en los ficheros `classification_incremental*.py`, `regression_incremental*.py` y `mnist.py` del directorio de `examples`.

A continuación presentamos algunos extractos de la salida de estos programas y un resumen de las pruebas de este apartado en la tabla 3.9.

Programa	Conjunto de datos	P	N_g	P'	Métrica global	Métrica incremental
<code>classification_incremental</code>	Dry Bean	1000	100	-	90.30 %	90.30 %
<code>classification_incremental_multiple</code>	Dry Bean	1000	100	50	90.40 %	90.45 %
<code>regression_incremental</code>	Carbon Nanotubes	1000	100	-	5.750e-5	5.750E-05
<code>regression_incremental_multiple</code>	Carbon Nanotubes	1000	100	50	5.750e-5	5.749E-05
<code>mnist</code>	MNIST-test	10	2	-	90.74 %	90.74 %

Tabla 3.9: Resumen de los resultados experimentales de agregación parcial

```

Run - fedheonn_ensemble
Run: classification_incremental
C:\Users\Abel\PycharmProjects\fedheonn_ensemble\venv\Scripts\python.exe
IID scenario
Training client: 1 of 1000 ( 0 - 8 ) - Classes: [ 2 3 4 5 6 ]
Training client: 2 of 1000 ( 9 - 17 ) - Classes: [ 2 3 4 5 6 ]
Training client: 3 of 1000 ( 19 - 27 ) - Classes: [ 0 2 3 5 6 ]
Training client: 4 of 1000 ( 28 - 36 ) - Classes: [ 1 2 3 4 5 6 ]
Training client: 5 of 1000 ( 38 - 46 ) - Classes: [ 0 1 2 3 4 5 6 ]
Training client: 6 of 1000 ( 47 - 55 ) - Classes: [ 0 3 4 5 6 ]
Training client: 7 of 1000 ( 57 - 65 ) - Classes: [ 0 3 4 5 6 ]
Training client: 8 of 1000 ( 66 - 74 ) - Classes: [ 0 2 3 4 5 6 ]
Training client: 9 of 1000 ( 76 - 84 ) - Classes: [ 0 1 2 3 4 5 ]
Training client: 10 of 1000 ( 85 - 93 ) - Classes: [ 0 3 4 5 6 ]
Training client: 11 of 1000 ( 95 - 103 ) - Classes: [ 0 2 3 4 5 ]
Training client: 12 of 1000 ( 104 - 112 ) - Classes: [ 1 2 3 4 5 6 ]
Training client: 13 of 1000 ( 114 - 122 ) - Classes: [ 0 2 3 4 ]
Training client: 14 of 1000 ( 123 - 131 ) - Classes: [ 1 2 3 5 6 ]
Training client: 15 of 1000 ( 133 - 141 ) - Classes: [ 3 4 5 6 ]
Training client: 16 of 1000 ( 142 - 150 ) - Classes: [ 0 4 5 6 ]
Training client: 17 of 1000 ( 152 - 160 ) - Classes: [ 2 3 4 5 6 ]
Training client: 18 of 1000 ( 161 - 169 ) - Classes: [ 0 3 4 5 6 ]
Training client: 19 of 1000 ( 171 - 179 ) - Classes: [ 3 4 5 6 ]
Training client: 20 of 1000 ( 181 - 189 ) - Classes: [ 0 2 3 5 6 ]
Training client: 21 of 1000 ( 190 - 198 ) - Classes: [ 0 1 3 5 6 ]
Training client: 22 of 1000 ( 200 - 208 ) - Classes: [ 0 1 2 3 4 ]
Training client: 23 of 1000 ( 209 - 217 ) - Classes: [ 1 2 4 5 6 ]
Training client: 24 of 1000 ( 219 - 227 ) - Classes: [ 0 3 4 6 ]
Training client: 25 of 1000 ( 228 - 236 ) - Classes: [ 2 3 4 5 6 ]
Training client: 26 of 1000 ( 238 - 246 ) - Classes: [ 2 3 4 6 ]
Training client: 27 of 1000 ( 247 - 255 ) - Classes: [ 0 2 3 5 6 ]
Training client: 28 of 1000 ( 257 - 265 ) - Classes: [ 0 2 3 4 5 ]
Training client: 29 of 1000 ( 266 - 274 ) - Classes: [ 0 3 4 5 6 ]
Training client: 30 of 1000 ( 276 - 284 ) - Classes: [ 1 2 3 4 5 6 ]
Training client: 31 of 1000 ( 285 - 293 ) - Classes: [ 0 3 4 6 ]
Training client: 32 of 1000 ( 295 - 303 ) - Classes: [ 0 2 3 4 6 ]
Training client: 33 of 1000 ( 304 - 312 ) - Classes: [ 0 2 3 4 5 6 ]
...
```
Run - fedheonn_ensemble
Run: classification_incremental
Training client: 777 of 1000 (7400 - 7470) - Classes: [1 2 3 4 5 6]
Training client: 998 of 1000 (9498 - 9506) - Classes: [0 2 3 4 5 6]
Training client: 999 of 1000 (9507 - 9515) - Classes: [0 2 3 5 6]
Training client: 1000 of 1000 (9517 - 9525) - Classes: [1 2 3 4 5 6]
Grouping clients: (100:100)
Grouping clients: (200:200)
Grouping clients: (300:300)
Grouping clients: (400:400)
Grouping clients: (500:600)
Grouping clients: (600:700)
Grouping clients: (700:800)
Grouping clients: (800:900)
Grouping clients: (900:1000)
Test accuracy after incremental group 1: 88.25 %
Test accuracy after incremental group 2: 88.52 %
Test accuracy after incremental group 3: 89.45 %
Test accuracy after incremental group 4: 89.72 %
Test accuracy after incremental group 5: 89.79 %
Test accuracy after incremental group 6: 89.96 %
Test accuracy after incremental group 7: 90.23 %
Test accuracy after incremental group 8: 90.21 %
Test accuracy after incremental group 9: 90.38 %
Test accuracy after incremental group 10: 90.38 %
Test accuracy global: 90.38 %
Test accuracy incremental: 90.38 %
Comparing W_glb[0] with W_inc[0]: OK
Comparing W_glb[1] with W_inc[1]: OK
Comparing W_glb[2] with W_inc[2]: OK
Comparing W_glb[3] with W_inc[3]: OK
Comparing W_glb[4] with W_inc[4]: OK
Comparing W_glb[5] with W_inc[5]: OK
Comparing W_glb[6] with W_inc[6]: OK
Process finished with exit code 0

```

Figura 3.38: Salida de classification\_incremental.py

```

Run - fedheonn_ensemble
Run: classification_incremental_multiple
C:\Users\Abel\PycharmProjects\fedheonn_ensemble\venv\Scripts\python.exe
IID scenario
Training client: 1 of 1000 (0 - 8)
Training client: 2 of 1000 (9 - 17)
Training client: 3 of 1000 (19 - 27)
Training client: 4 of 1000 (28 - 36)
Training client: 5 of 1000 (38 - 46)
Training client: 6 of 1000 (47 - 55)
Training client: 7 of 1000 (57 - 65)
Training client: 8 of 1000 (66 - 74)
Training client: 9 of 1000 (76 - 84)
Training client: 10 of 1000 (85 - 93)
Training client: 11 of 1000 (95 - 103)
Training client: 12 of 1000 (104 - 112)
Training client: 13 of 1000 (114 - 122)
Training client: 14 of 1000 (123 - 131)
Training client: 15 of 1000 (133 - 141)
Training client: 16 of 1000 (142 - 150)
Training client: 17 of 1000 (152 - 160)
Training client: 18 of 1000 (161 - 169)
Training client: 19 of 1000 (171 - 179)
Training client: 20 of 1000 (181 - 189)
Training client: 21 of 1000 (190 - 198)
Training client: 22 of 1000 (200 - 208)
Training client: 23 of 1000 (209 - 217)
Training client: 24 of 1000 (219 - 227)
Training client: 25 of 1000 (228 - 236)
Training client: 26 of 1000 (238 - 246)
Training client: 27 of 1000 (247 - 255)
Training client: 28 of 1000 (257 - 265)
Training client: 29 of 1000 (266 - 274)
Training client: 30 of 1000 (276 - 284)
Training client: 31 of 1000 (285 - 293)
Training client: 32 of 1000 (295 - 303)
Training client: 33 of 1000 (304 - 312)
...
```
Run - fedheonn_ensemble
Run: classification_incremental_multiple
Training client: 41 of 50 ( 7621 - 7810 )
Training client: 42 of 50 ( 7812 - 8001 )
Training client: 43 of 50 ( 8002 - 8191 )
Training client: 44 of 50 ( 8193 - 8382 )
Training client: 45 of 50 ( 8383 - 8572 )
Training client: 46 of 50 ( 8574 - 8763 )
Training client: 47 of 50 ( 8764 - 8953 )
Training client: 48 of 50 ( 8955 - 9144 )
Training client: 49 of 50 ( 9145 - 9334 )
Training client: 50 of 50 ( 9336 - 9525 )
Test accuracy after incremental group 1: 90.25 %
Test accuracy after incremental group 2: 90.30 %
Test accuracy after incremental group 3: 90.35 %
Test accuracy after incremental group 4: 90.43 %
Test accuracy after incremental group 5: 90.45 %
Test accuracy after incremental group 6: 90.45 %
Test MSE global: 90.40 %
Test MSE incremental: 90.45 %
Comparing W_glb[0] with W_inc[0]: KO
DIFF %: ['143.57%', '5.98%', '4.59%', '13.27%', '47.12%', '4.40%']
Comparing W_glb[1] with W_inc[1]: KO
DIFF %: ['0.04%', '5.20%', '4.68%', '6.68%', '107.78%', '1.03%']
Comparing W_glb[2] with W_inc[2]: KO
DIFF %: ['2.35%', '126.45%', '12.88%', '99.71%', '177.88%', '14.2']
Comparing W_glb[3] with W_inc[3]: KO
DIFF %: ['0.60%', '0.52%', '2.45%', '0.07%', '1.16%', '0.63%']
Comparing W_glb[4] with W_inc[4]: KO
DIFF %: ['71.82%', '15.86%', '22.40%', '13.54%', '11.25%', '3.29%']
Comparing W_glb[5] with W_inc[5]: KO
DIFF %: ['17.98%', '0.83%', '4.72%', '1.78%', '16.07%', '14.51%']
Comparing W_glb[6] with W_inc[6]: KO
DIFF %: ['2.29%', '1.55%', '1.77%', '3.14%', '0.05%', '2.76%']
Process finished with exit code 0

```

Figura 3.39: Salida de classification_incremental_multiple.py

```

Run - fedheonn_ensemble
Run: regression_incremental <...>
C:\Users\Abel\PycharmProjects\fedheonn_ensemble\venv\Scripts\python.exe
Training client: 1 of 1000 ( 0 - 6 )
Training client: 2 of 1000 ( 7 - 13 )
Training client: 3 of 1000 ( 15 - 21 )
Training client: 4 of 1000 ( 22 - 28 )
Training client: 5 of 1000 ( 30 - 36 )
Training client: 6 of 1000 ( 37 - 43 )
Training client: 7 of 1000 ( 45 - 51 )
Training client: 8 of 1000 ( 52 - 58 )
Training client: 9 of 1000 ( 60 - 66 )
Training client: 10 of 1000 ( 67 - 73 )
Training client: 11 of 1000 ( 75 - 81 )
Training client: 12 of 1000 ( 82 - 88 )
Training client: 13 of 1000 ( 90 - 96 )
Training client: 14 of 1000 ( 97 - 103 )
Training client: 15 of 1000 ( 105 - 111 )
Training client: 16 of 1000 ( 112 - 118 )
Training client: 17 of 1000 ( 120 - 126 )
Training client: 18 of 1000 ( 127 - 133 )
Training client: 19 of 1000 ( 135 - 141 )
Training client: 20 of 1000 ( 142 - 148 )
Training client: 21 of 1000 ( 150 - 156 )
Training client: 22 of 1000 ( 157 - 163 )
Training client: 23 of 1000 ( 165 - 171 )
Training client: 24 of 1000 ( 172 - 178 )
Training client: 25 of 1000 ( 180 - 186 )
Training client: 26 of 1000 ( 187 - 193 )
Training client: 27 of 1000 ( 195 - 201 )
Training client: 28 of 1000 ( 202 - 208 )
Training client: 29 of 1000 ( 210 - 216 )
Training client: 30 of 1000 ( 217 - 223 )
Training client: 31 of 1000 ( 225 - 231 )
Training client: 32 of 1000 ( 232 - 238 )
Training client: 33 of 1000 ( 240 - 246 )
Training client: 34 of 1000 ( 247 - 253 )
...
Training client: 982 of 1000 ( 7801 - 7807 )
Training client: 983 of 1000 ( 7368 - 7374 )
Training client: 984 of 1000 ( 7376 - 7382 )
Training client: 985 of 1000 ( 7383 - 7389 )
Training client: 986 of 1000 ( 7391 - 7397 )
Training client: 987 of 1000 ( 7398 - 7404 )
Training client: 988 of 1000 ( 7406 - 7412 )
Training client: 989 of 1000 ( 7413 - 7419 )
Training client: 990 of 1000 ( 7421 - 7427 )
Training client: 991 of 1000 ( 7428 - 7434 )
Training client: 992 of 1000 ( 7436 - 7442 )
Training client: 993 of 1000 ( 7443 - 7449 )
Training client: 994 of 1000 ( 7451 - 7457 )
Training client: 995 of 1000 ( 7458 - 7464 )
Training client: 996 of 1000 ( 7466 - 7472 )
Training client: 997 of 1000 ( 7473 - 7479 )
Training client: 998 of 1000 ( 7481 - 7487 )
Training client: 999 of 1000 ( 7488 - 7494 )
Training client: 1000 of 1000 ( 7496 - 7502 )
Test accuracy after incremental group 1: 0.00005759
Test accuracy after incremental group 2: 0.00005765
Test accuracy after incremental group 3: 0.00005759
Test accuracy after incremental group 4: 0.00005758
Test accuracy after incremental group 5: 0.00005757
Test accuracy after incremental group 6: 0.00005756
Test accuracy after incremental group 7: 0.00005754
Test accuracy after incremental group 8: 0.00005753
Test accuracy after incremental group 9: 0.00005750
Test MSE global: 0.00005750
Test MSE incremental: 0.00005750
Comparing W_glb[0] with W_inc[0]: OK
Comparing W_glb[1] with W_inc[1]: OK
Comparing W_glb[2] with W_inc[2]: OK
Process finished with exit code 0
|
```

Figura 3.40: Salida de regression_incremental.py

```

Run - fedheonn_ensemble
Run: regression_incremental_multiple <...>
C:\Users\Abel\PycharmProjects\fedheonn_ensemble\venv\Scripts\python.exe
Training client: 1 of 1000 ( 0 - 6 )
Training client: 2 of 1000 ( 7 - 13 )
Training client: 3 of 1000 ( 15 - 21 )
Training client: 4 of 1000 ( 22 - 28 )
Training client: 5 of 1000 ( 30 - 36 )
Training client: 6 of 1000 ( 37 - 43 )
Training client: 7 of 1000 ( 45 - 51 )
Training client: 8 of 1000 ( 52 - 58 )
Training client: 9 of 1000 ( 60 - 66 )
Training client: 10 of 1000 ( 67 - 73 )
Training client: 11 of 1000 ( 75 - 81 )
Training client: 12 of 1000 ( 82 - 88 )
Training client: 13 of 1000 ( 90 - 96 )
Training client: 14 of 1000 ( 97 - 103 )
Training client: 15 of 1000 ( 105 - 111 )
Training client: 16 of 1000 ( 112 - 118 )
Training client: 17 of 1000 ( 120 - 126 )
Training client: 18 of 1000 ( 127 - 133 )
Training client: 19 of 1000 ( 135 - 141 )
Training client: 20 of 1000 ( 142 - 148 )
Training client: 21 of 1000 ( 150 - 156 )
Training client: 22 of 1000 ( 157 - 163 )
Training client: 23 of 1000 ( 165 - 171 )
Training client: 24 of 1000 ( 172 - 178 )
Training client: 25 of 1000 ( 180 - 186 )
Training client: 26 of 1000 ( 187 - 193 )
Training client: 27 of 1000 ( 195 - 201 )
Training client: 28 of 1000 ( 202 - 208 )
Training client: 29 of 1000 ( 210 - 216 )
Training client: 30 of 1000 ( 217 - 223 )
Training client: 31 of 1000 ( 225 - 231 )
Training client: 32 of 1000 ( 232 - 238 )
Training client: 33 of 1000 ( 240 - 246 )
Training client: 34 of 1000 ( 247 - 253 )
...
Training client: 33 of 50 ( 4802 - 4951 )
Training client: 34 of 50 ( 4952 - 5101 )
Training client: 35 of 50 ( 5102 - 5251 )
Training client: 36 of 50 ( 5252 - 5401 )
Training client: 37 of 50 ( 5402 - 5551 )
Training client: 38 of 50 ( 5552 - 5701 )
Training client: 39 of 50 ( 5703 - 5852 )
Training client: 40 of 50 ( 5853 - 6002 )
Training client: 41 of 50 ( 6003 - 6152 )
Training client: 42 of 50 ( 6153 - 6302 )
Training client: 43 of 50 ( 6303 - 6452 )
Training client: 44 of 50 ( 6453 - 6602 )
Training client: 45 of 50 ( 6603 - 6752 )
Training client: 46 of 50 ( 6753 - 6902 )
Training client: 47 of 50 ( 6903 - 7052 )
Training client: 48 of 50 ( 7053 - 7202 )
Training client: 49 of 50 ( 7203 - 7352 )
Training client: 50 of 50 ( 7353 - 7502 )
Test accuracy after incremental group 1: 0.00005801
Test accuracy after incremental group 2: 0.00005753
Test accuracy after incremental group 3: 0.00005752
Test accuracy after incremental group 4: 0.00005752
Test accuracy after incremental group 5: 0.00005751
Test accuracy after incremental group 6: 0.00005749
Test MSE global: 0.00005750
Test MSE incremental: 0.00005749
Comparing W_glb[0] with W_inc[0]: KO
DIFF %: ['0.00%', '114.53%', '595.01%', '0.01%', '1.54%', '3.93%']
Comparing W_glb[1] with W_inc[1]: KO
DIFF %: ['0.00%', '523.37%', '120.53%', '4.02%', '0.00%', '1.81%']
Comparing W_glb[2] with W_inc[2]: KO
DIFF %: ['0.01%', '51.60%', '21.77%', '1.21%', '48.95%', '0.01%']
Process finished with exit code 0
|
```

Figura 3.41: Salida de regression_incremental_multiple.py

```

Run - fedheonn_ensemble
Run: mnist ×
C:\Users\Abel\PycharmProjects\fedheonn_ensemble\venv\Scripts\python.exe
INFO    - [*] MNIST DIGITS DATASET (1797 samples) [*]
INFO    - Data normalization (z-score)
INFO    - IID scenario
INFO    - Starting timer to [main]
INFO    - Training client: 1 of 10 (0-124)
INFO    - Training client: 2 of 10 (125-249)
INFO    - Training client: 3 of 10 (251-375)
INFO    - Training client: 4 of 10 (377-501)
INFO    - Training client: 5 of 10 (502-626)
INFO    - Training client: 6 of 10 (628-752)
INFO    - Training client: 7 of 10 (754-878)
INFO    - Training client: 8 of 10 (879-1003)
INFO    - Training client: 9 of 10 (1005-1129)
INFO    - Training client: 10 of 10 (1131-1255)
INFO    - Grouping clients: (0:2)
INFO    - Grouping clients: (2:4)
INFO    - Grouping clients: (4:6)
INFO    - Grouping clients: (6:8)
INFO    - Grouping clients: (8:10)
INFO    - Test accuracy global: 90.74 %
INFO    - Test accuracy incremental: 90.74 %
INFO    - Elapsed time running [main]: [3.048] seconds

Process finished with exit code 0

```

Figura 3.42: Salida de `mnist.py`

En la tabla 3.9 las métricas (columnas *Métrica global* y *Métrica incremental*) para las pruebas de clasificación corresponden a la exactitud, en %, y el MSE para las de regresión.

En los resultados correspondientes a *classification_incremental* y *regression_incremental*, figuras 3.38 y 3.40, podemos observar como, para un mismo número de clientes P , la agregación global de todos ellos por separado, y la agregación parcial e incremental de estos en grupos de N_g clientes, repercuten en métricas y pesos equivalentes. Este es también el caso de la última prueba, `mnist`, correspondiente a la figura 3.42.

En cambio, en las pruebas con diferente número de clientes, correspondientes a *classification_incremental_multiple* y *regression_incremental_multiple*, figuras 3.39 y 3.41, la agregación global de los primeros P clientes y la incremental de los P' segundos (en grupos aleatorios), no resultan en exactamente las mismas métricas, aún siendo bastante parecidas. Esto es debido a que, como el mismo dataset D está repartido entre un número distinto de clientes ($P \gg P'$), a su vez agrupados de distinto modo, el compendio de la información sobre el que entran unos y otros es distinto, repercutiendo en matrices $\llbracket m_p \rrbracket$, US_p semejantes pero no exactamente iguales.

En definitiva, siempre que el número de clientes P sea idéntico **se alcanzan matemáticamente los mismos resultados** tanto para en el algoritmo original de agregación y cálculo de pesos global como en los nuevos procesos independientes de agregación parcial en lotes y cálculo de pesos óptimos.

3.7.2. Ensamblaje

La experimentación relativa a la implementación de la técnica de ensamblaje conocida como *Random Patches* [23], descrita en apartados anteriores, constaba de tres puntos: **primero**, probar que puede conseguir mejores rendimientos en algunos casos, **segundo**, analizar la sensibilidad de sus hiperparámetros sobre la exactitud para varios conjuntos distintos y, **tercero**, comparar la eficiencia de este modelo ampliado contra el original. Este último punto se analizará en la sección 3.7.5, englobado con otros análisis energéticos, además de comparaciones con otro tipo de modelos.

Comenzamos la experimentación con el **análisis de sensibilidad** sobre distintos conjuntos de datos, con respecto a la exactitud, realizando una búsqueda en cuadrícula tipo *grid search*, efectuando una validación cruzada de tipo K-fold con 10 partes o *splits* para cada serie de hiperparámetros ($T, \rho_s, \beta_s, \rho_f, \beta_f$), resultando en una **exactitud media** y **desviación típica** generada por cada validación. Los conjuntos de datos usados, los rangos paramétricos seleccionados y el espacio de búsqueda total se detallan en la tabla 3.43.

λ	T	ρ_s	ρ_f	$\beta_s - \beta_f$
0.01	2	0.05	0.5	FALSE
	5	0.1	0.7	TRUE
	10	0.15	0.9	
	20			
MINIBOONE		TOTAL: 144		
1	2	0.1	0.7	FALSE
	5	0.2	0.8	TRUE
	10	0.3	0.9	
		0.4		
		0.5		
SKIN		TOTAL: 360		
0.01	25	0.2	0.7	FALSE
	50	0.3	0.8	TRUE
	75	0.4	0.9	
	100	0.5	1	
		0.6		
MNIST		TOTAL: 640		
0.01	2	0.1	0.1	FALSE
	5	0.2	0.2	TRUE
	10	0.3	0.3	
	25	0.4	0.4	
	50	0.5	0.5	
	75	0.6	0.6	
	100	0.7	0.7	
	200	0.8	0.8	
		0.9	0.9	
	1	1		
CARBON - DRYBEAN		TOTAL: 12800		

Figura 3.43: Tabla con el rango paramétrico para los distintos *grid-search* ejecutados

Los programas diseñados para realizar este *cross-validation grid-search* se pueden encontrar bajo el nombre *_gridsearchcv_remote.py en el directorio examples y exportaron sus respectivos resultados a un fichero tipo Excel, de los que dejamos una copia bajo ese mismo directorio.

La longitud en la dimensión de los distintos espacios paramétricos es inversamente proporcional al tiempo de ejecución necesario para una iteración sobre los distintos conjuntos, de modo que; en aquellos con entrenamientos más pesados, como *MiniBoone*, *Skin* o *MNIST*, se realizó un barrido previo más tosco para afinar el rango de valores adecuado de la búsqueda paramétrica.

En la sección 4.4 del anexo de resultados dejamos unas imágenes con la salida de estos programas, cuya ejecución tardó desde unas horas hasta un par de días (figuras 4.5 a 4.7), seguido de unas tablas con los resultados alcanzados para cada conjunto de datos, extraídos de los ficheros Excel exportados por los programas previos, y de las cuales se dirimen unas primeras conclusiones de este apartado (tablas 4.9 a 4.10):

- La **efectividad de esta técnica** en la exactitud es totalmente dependiente del conjunto de datos y estimador base utilizado (*refiriéndonos al modelo sobre el que aplica*).
- Existen conjuntos de datos en los que su implementación **no repercute en ninguna mejora** apreciable y, por tanto, puede ser descartada.

Las métricas óptimas obtenidas y los hiperparámetros correspondientes, tanto para la versión original del modelo, como para la versión que incorpora el ensamblaje, se presentan en la tabla 3.10.

Conjunto de datos ⁶	λ (reg.)	f.act.	T	ρ_s	β_s	ρ_f	β_f	Exactitud o MSE (μ, σ)
MNIST-test	0.01	logs	-	-	-	-	-	88.15 % 3.14 %
MNIST-test	0.01	logs	75	0.2	✓	0.8	X	95.94 % 2.37 %
Skin	10	logs	-	-	-	-	-	80.52 % 0.31 %
Skin	10	logs	2	0.1	X	0.7	✓	84.21 % 0.34 %
MiniBoone	0.01	logs	-	-	-	-	-	85.34 % 0.30 %
MiniBoone	0.01	logs	2	0.05	X	0.9	X	86.85 % 0.33 %
Dry Bean	0.01	logs	-	-	-	-	-	90.69 % 1.01 %
Dry Bean	0.01	logs	10	0.8	✓	1	X	91.01 % 1.03 %
Carbon Nanotubes	0.01	linear	-	-	-	-	-	7.85E-05 8.70E-05
Carbon Nanotubes	0.01	linear	75	0.4	✓	1	X	7.83E-05 8.69E-05

Tabla 3.10: Resultados del modelo con ensamblaje con los hiperparámetros óptimos obtenidos con validación cruzada

Como se puede apreciar, las diferencias en algunos casos son significativas, destacando una **mejora** de casi un **8 %** en la exactitud para el caso del conjunto de datos *MNIST-test* y casi un **4 %** para el conjunto *Skin*, de las versiones con ensamblaje respecto a las originales, usando ambos los mismos hiperparámetros (a excepción de los propios del *ensemble*). En el resto de conjuntos la mejora es más discreta, del 1 y 0.5 % respectivamente para la exactitud

⁶Las filas con conjuntos y resultados en negrita representan la versión con ensamblaje.

en *MiniBoone* y *Dry Bean*, y una reducción de 0.02×10^{-5} , o un 0.25 % relativo, para el MSE de *Carbon Nanotubes*.

Los hiperparámetros propios del ensamblaje tipo *Random Patches* de la tabla 3.10 usados han sido aquellos que ofrecieron mejores métricas para cada conjunto, luego de realizar la búsqueda paramétrica detallada al inicio, que, aún sin ser demasiado exhaustiva, da pie a observar que:

- Un alto número de estimadores base T no tiene por qué resultar en mejores métricas.
- Se abre un amplio espacio de búsqueda nuevo, para dar con el ajuste más eficaz y/o eficiente de este tipo de ensamblaje.

3.7.3. Paralelismo

De forma muy parecida, las pruebas de este apartado se centrarán en **demostrar** que la parallelización del proceso, tanto en cliente como coordinador, es capaz de reducir los tiempos globales de ejecución y explorar, brevemente, como afectan ciertos hiperparámetros del ensamblaje, como el número de estimadores base T o el tamaño relativo del parche $\rho_s \times \rho_f$ a dicha reducción.

No merece la pena realizar estudios mayores – en primera instancia – ya que el principal factor en tal ahorro tiene también mucho que ver con las dimensiones del conjunto de entrenamiento y el número de clientes involucrados.

En esta sección nos centraremos específicamente en el dataset *MNIST-test* para todos los ejemplos siguientes, con un número de clientes P de 4 y agrupados en lotes N_g de 2 clientes.

Para evidenciar el primer punto ejecutamos el programa `mnist_parallel.py`, localizado en la carpeta `examples`, donde se corre el entrenamiento global e incremental del modelo bajo un ensamblaje de $T = 50$ estimadores base y tamaño relativo del parche $(\rho_s \times \rho_f) = 1$; **con y sin paralelización**.

En las figuras 4.8 y 4.9 del anexo de resultados mostramos la salida de estos programas y en la tabla 3.11 recopilamos sus resultados. En esta, hemos desglosado y cronometrado los tiempos de ejecución de las distintas tareas involucradas en el proceso de entrenamiento conjunto del modelo de la siguiente forma:

- Para la lista de clientes, recordemos $P = 4$, medimos el tiempo de entrenamiento de cada conjunto de datos local (`fit`).
- Para el coordinador central, la agregación y cálculo de pesos óptimos se realiza de modo *global*, como en la versión original, y de modo *incremental*, en lotes de $N_g = 2$, realizando primero la agregación parcial y, por último, el cálculo de pesos óptimos, en procesos separados.
- También hemos incluido el tiempo requerido para serializar el contexto y vectores encriptados, previo a su procesamiento en paralelo, necesario para las tareas de agregación (`aggregate`) y cálculo de pesos óptimos (`calc_weights`), representando el preprocesamiento de estos casos (`pre-proc`).

Se puede observar una **reducción significativa** en los tiempos de **entrenamiento** que concierne a los **clientes** de la red, siendo hasta 1/3 más rápido para todos los casos, pasando de un `fit` global de los clientes de 98 (s) a 62 (s), por ejemplo.

Desglose de tiempos de ejecución (s) EN SERIE

	Clientes		Coordinador	
	FIT	GLOBAL	INCREMENTAL	
CLIENTE 1:	23.29	PRE-PROC:	0.00	PRE-PROC: 0.00
CLIENTE 2:	24.65	AGGREGATE:	715.18	AGG_PARTIAL 1: 2.90
CLIENTE 3:	24.76	-		AGG_PARTIAL 2: 5.26
CLIENTE 4:	24.89	-		CALC_WEIGHTS: 728.04
Σ :	97.59	Σ :	716.57	Σ : 737.53

EN PARALELO

	Clientes		Coordinador	
	FIT	GLOBAL	INCREMENTAL	
CLIENTE 1:	14.92	PRE-PROC:	30.10	PRE-PROC: 9.66
CLIENTE 2:	14.86	AGGREGATE:	728.46	AGG_PARTIAL 1: 7.51
CLIENTE 3:	16.22	-		AGG_PARTIAL 2: 7.10
CLIENTE 4:	16.13	-		CALC_WEIGHTS: 692.22
Σ :	62.13	Σ :	730.14	Σ : 707.58

Tabla 3.11: Tiempos de ejecución de los distintos procesos involucrados en el entrenamiento conjunto de FedHEONN

Por la contra, la **paralelización en el coordinador no resulta tan efectiva**. Cuando se efectúa una *agregación global* (algoritmo original del modelo) con este ensamblaje, el proceso tarda más incluso que sin la paralelización, debido, en mayor parte, al gran coste añadido que supone el preprocessado de datos (serialización y deserialización de contexto y vectores) y creación y manejo de procesos necesarios para ello. Por ejemplo, medio minuto, 30 (s) de los 728 (s) de dicha agregación se destinaron únicamente a serializar y preparar en lotes los datos necesarios para cada proceso del *pool*.

El mismo problema se aprecia también en la nueva versión, denominada *incremental*, donde separamos la agregación de los datos del cálculo de pesos óptimos; aquí, la **paralelización de la primera (agg_partial) tarda más en ejecutarse** que su equivalente en serie (debido al *sobrecoste* del manejo de los procesos), pero, por la contra, el **proceso de cálculo de pesos óptimos (calculate_weights) si que se redujo, aproximadamente, en un 5%**, de 728 (s) a 692 (s).

En las gráficas de la figura 3.44, creadas con el programa `mnist_parallel_tests.py` del directorio de `examples`, representamos los tiempos de ejecución de los distintos procesos del modelo vistos anteriormente; específicamente, el *fit* de los clientes y la *agregación parcial* y *cálculo de pesos óptimos* del coordinador, según el número de estimadores base T empleados, **en serie y en paralelo**.

De estas podemos extraer varias conclusiones:

- La paralelización implementada es **efectiva en el entrenamiento de los clientes**

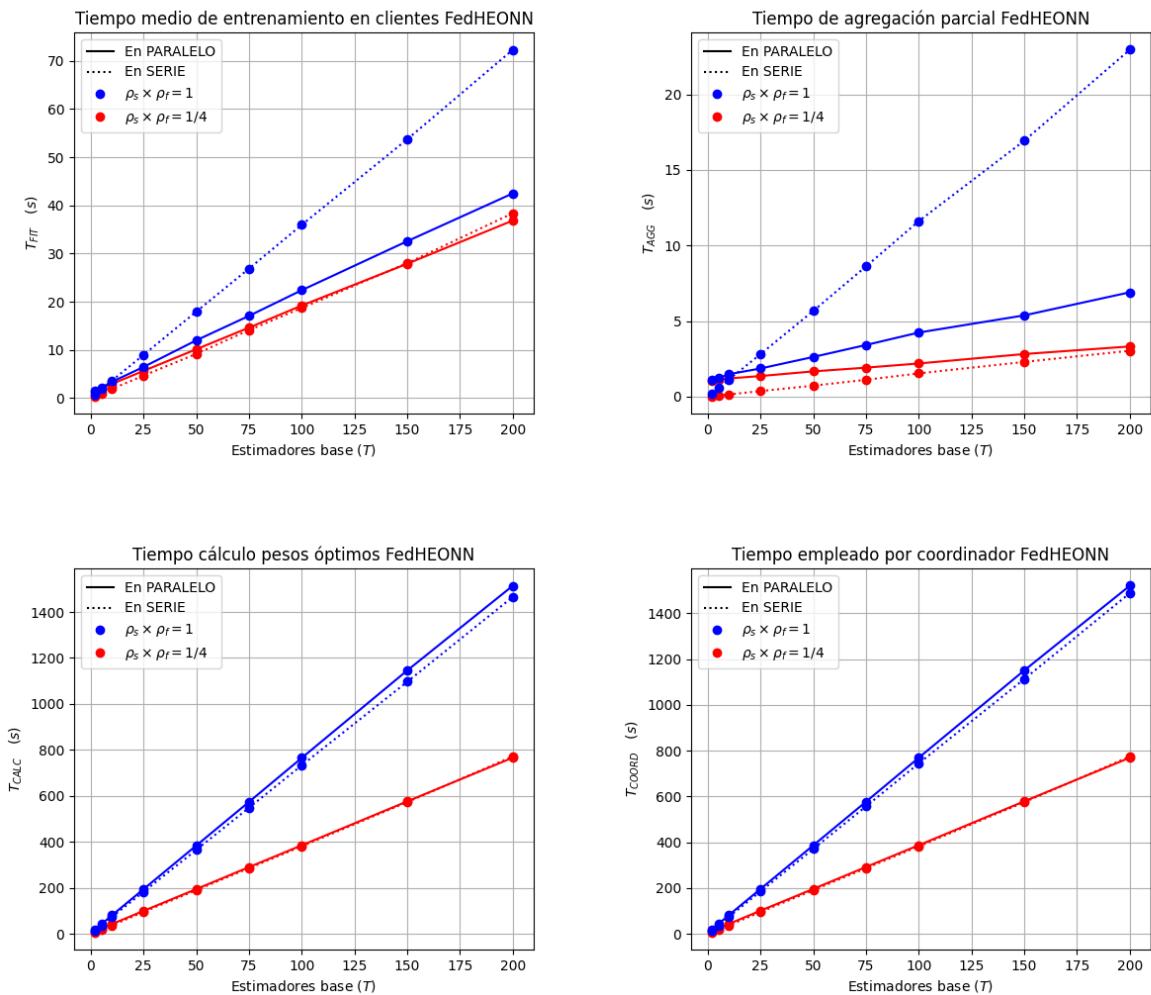


Figura 3.44: Tiempos para las distintas tareas involucradas en FedHEONN

(fit) y la **agregación parcial del coordinador** (aggregate_partial), pero no lo es (por lo menos para el rango de estimadores base estudiados) en el cálculo de pesos óptimos (calculate_weights).

- En la parallelización del cálculo de pesos óptimos, aún repartiendo el trabajo entre varios cores, el sobrecoste de serializar y manipular vectores encriptados, realizar operaciones con ellos y crear *pools* de procesos hace simplemente que la **implementación desarrollada para esta tarea no sea útil**.
- Como, a mayores, se trata de la tarea más lenta, la parallelización del coordinador apenas muestra alguna diferencia importante frente a la versión en serie ($T_{COORD} = T_{AGG} + T_{CALC}$).
- En cambio, para clientes con poca potencia computacional pero multi-núcleo, como algunos dispositivos embebidos, si que supone una ventaja importante frente a la versión original.
- La parallelización funcionó bien para aquellos procesos que parallelizaron las operaciones costosas sobre *datos-en-claro*, por ejemplo la descomposición SVD, necesario en el fit y

- aggregate_partial, y **no en donde hubo que trabajar con vectores encriptados**.
- Los resultados anteriores sugieren que la paralelización del entrenamiento en los clientes es favorable para todo T , mientras que la de la agregación parcial en el coordinador, solo lo es en casos con alto número de estimadores base T y/o gran tamaño relativo $\rho_s \times \rho_f$, como se aprecia en la figura superior derecha de 3.44, dando pie a un umbral – por determinar y específico de cada dataset –, a partir del cual compensa esta paralelización.

3.7.4. Prototipo desarrollado para el entorno federado

La demostración realizada consistió en un portátil (Lenovo Y540) actuando como servidor/coordinador, con el otro (Lenovo X200) ejecutando dos clientes, todos conectados a la misma red doméstica.

Uno de estos clientes entrenará con la mitad de los datos de entrenamiento del dataset X_p y el otro con la parte restante. El primero que finalice su entrenamiento enviará su información al servidor, donde el coordinador la agregará y encolará la parte restante recibida mientras tanto. Una vez realizada la agregación incremental de ambas, y con la cola ya vacía, se calcularán los pesos óptimos, que se enviarán a uno de los clientes para que este realice una predicción sobre el conjunto de datos de *test*.

La **demo** no se pudo realizar involucrando a los dispositivos Raspberry y Jetson Nano, tal y como se había planteado al principio, debido a que la librería necesaria para la encriptación homomórfica TenSEAL no está soportada bajo las arquitecturas ARM disponibles. En un futuro se espera que se soporten, pero de momento prescindimos de ellos en la red federada.

A continuación presentamos unas figuras mostrando diferentes momentos del entrenamiento federado, explicando brevemente qué pasa en cada una de ellas:

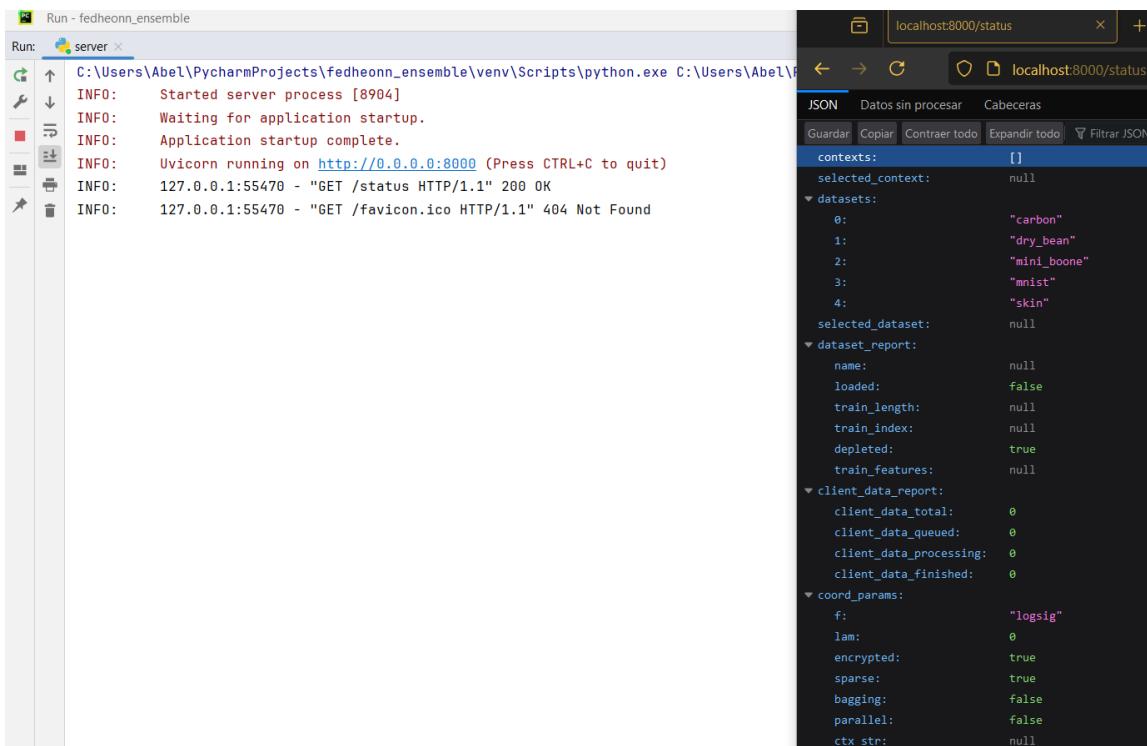


Figura 3.45: Arranque del servidor.

```
demo_config.ini x
1 #Configuration file for dataset
2 #Prefix indicates data-type: str.
3
4 #Dataset selection
5 [dataset]
6 str_selected_dataset = mnist
7
8 #Server IP/Port
9 [server]
10 str_hostname = 192.168.0.14
11 int_port = 8000
12
13 #Algorithm tuning: common and cl.
14 [fedheonn]
15 str_activation_function = logs
16 bol_encrypted = True
17 bol_sparse = True
18 bol_ensemble = True
19
20 [fedheonn_coord]
21 flt_regularization = 0.1
22 bol_parallel = False
23
24 [fedheonn_client]
25 bol_classification = True
26 int_n_estimators = 8
27 flt_p_samples = 1.0
28 bol_b_samples = True
29 flt_p_features = 1.0
30 bol_b_features = False
31 bol_parallel = False
```

Figura 3.46: Configuración utilizada durante la *demo* para cliente y coordinador FedHEONN.

The terminal window shows the command `python examples/demo_server.py` being run. The application logs indicate the server has started and is listening on port 8000. It handles various requests, including dataset loading and context switching. To the right, a JSON status report is displayed, showing the current state of the system, including contexts, datasets, and client data reports.

```
(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_server.py
(venv) [escritorio@leba fedheonn_ensemble]$
```

```

{
  "contexts": {
    "0": "demo_server_context",
    "selected_context": "demo_server_context"
  },
  "datasets": {
    "0": "carbon",
    "1": "dry_beans",
    "2": "mini_boone",
    "3": "mnist",
    "4": "skin",
    "selected_dataset": "mnist"
  },
  "dataset_report": {
    "name": "mnist",
    "loaded": true,
    "train_length": 1257,
    "train_index": 0,
    "depleted": false,
    "train_features": 64
  },
  "client_data_report": {
    "client_data_total": 0,
    "client_data_queued": 0,
    "client_data_processing": 0,
    "client_data_finished": 0
  },
  "coord_params": {
    "f": "logsig",
    "lam": 0.1,
    "encrypted": true,
    "sparse": true,
    "bagging": true,
    "parallel": false,
    "ctx_str": null
  }
}

```

Figura 3.47: Ejecución de `demo_server.py` y reflejo en el `status` del servidor.

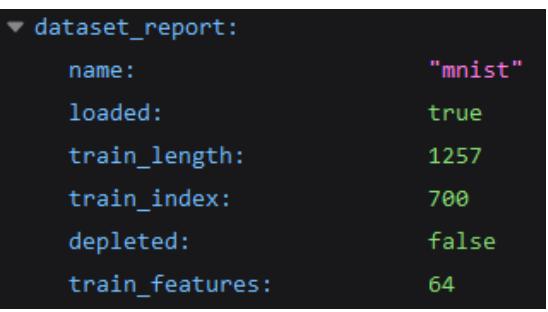
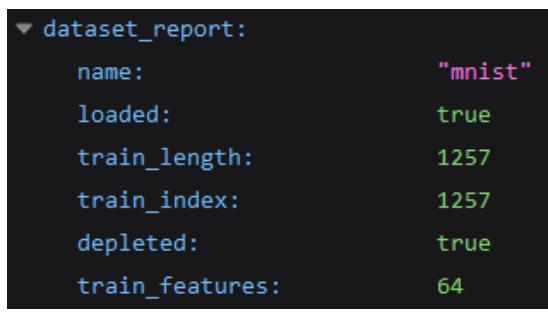
	
(a) Primer cliente, obteniendo 700 registros.	(b) Segundo cliente, recogiendo los restantes.

Figura 3.48: Descarga de los datos del conjunto de entrenamiento desde el servidor.

Hasta ahora hemos iniciado el servidor y configurado los parámetros del coordinador acoplado (figuras 3.45 y 3.46) mediante la ejecución de `server.py` en un portátil y de `demo_server.py` en el otro. A mayores, también se ha seleccionado el `dataset` así como generado y subido el contexto público de TenSEAL al servidor (figura 3.47).

Seguidamente, en el portátil secundario se iniciarán dos instancias de `demo_client.py`,

programas que empiezan descargando los datos de entrenamiento del servidor, como se observa en las figuras 3.48 y 3.49, hasta que el *dataset* se agota. Luego, empezamos el entrenamiento de ambos simultáneamente y esperamos a que, según acaben su *entrenamiento local*, manden esta información al servidor para su agregación y cálculo de pesos óptimos (figura 3.50).

Como se ve en la figura 3.51, el número de clientes conectados y procesados, así como la gestión de la cola de agregación, funciona como se había planteado en la sección 3.6.5. Por último, una vez finalizado el proceso de agregación y cálculo por parte del servidor, se reciben los pesos óptimos y se realiza una predicción sobre el conjunto de *test* (figura 3.52).

```
(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_client.py
Start training client? y
Training client...
Elapsed time: 8.69 s
Sending fitted train data...
[Progress Bar]
```

Figura 3.49: Arranque de los clientes.

```
(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_client.py
Start training client? y
Training client...
Elapsed time: 8.69 s
Sending fitted train data...
Data enqueued!: 9bfada6c-4e64-4625-b0d2-07e28444e482
Check queue status of partial data? (y/n): y
Successful: 'PROCESSING'
Check queue status of partial data? (y/n): 
[Progress Bar]

(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_client.py
Start training client? y
Training client...
Elapsed time: 11.42 s
Sending fitted train data...
Data enqueued!: 6f20bbcc-3bbc-4292-9390-11e714b5e84c
Check queue status of partial data? (y/n): y
Successful: 'ENQUEUED'
Check queue status of partial data? (y/n): 
[Progress Bar]
```

Figura 3.50: Comienzo del entrenamiento local en ambos clientes.

client_data_report:			
client_data_total:	client_data_queued:	client_data_processing:	client_data_finished:
2	1	1	0
2	0	1	1
2	0	0	2

(a) Agregando primer lote.

(b) Agregando segundo lote.

(c) Clientes procesados.

Figura 3.51: Funcionamiento de la cola de agregación en el servidor.

```

Terminal - Archivo Editar Ver Terminal Pestañas Ayuda
(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_client.py
      Start training client?: y
Training client...
Elapsed time: 8.69 s
Sending fitted train data...
Data enqueued!: 9bfada6c-4e64-4625-b0d2-07e28444e482
Check queue status of partial data? (y/n): y
Successful: 'PROCESSING'
Check queue status of partial data? (y/n): y
Successful: 'FINISHED'
Check queue status of partial data? (y/n): n
Receive weights and predict?: y
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.614] seconds
Accuracy on whole TEST data: 91.67
(venv) [escritorio@leba fedheonn_ensemble]$ 

Terminal - Archivo Editar Ver Terminal Pestañas Ayuda
(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_client.py
      Start training client?: y
Training client...
Elapsed time: 11.42 s
Sending fitted train data...
Data enqueued!: 6f20bbcc-3bbc-4292-9390-11e714b5e84c
Check queue status of partial data? (y/n): y
Successful: 'ENQUEUED'
Check queue status of partial data? (y/n): y
Successful: 'PROCESSING'
Check queue status of partial data? (y/n): n
Receive weights and predict?: y
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.614] seconds
Accuracy on whole TEST data: 91.67
(venv) [escritorio@leba fedheonn_ensemble]$ 

Terminal - Archivo Editar Ver Terminal Pestañas Ayuda
(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_client.py
      Start training client?: y
Training client...
Elapsed time: 8.69 s
Sending fitted train data...
Data enqueued!: 9bfada6c-4e64-4625-b0d2-07e28444e482
Check queue status of partial data? (y/n): y
Successful: 'PROCESSING'
Check queue status of partial data? (y/n): y
Successful: 'FINISHED'
Check queue status of partial data? (y/n): n
Receive weights and predict?: y
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.614] seconds
Accuracy on whole TEST data: 91.67
(venv) [escritorio@leba fedheonn_ensemble]$ 

Terminal - Archivo Editar Ver Terminal Pestañas Ayuda
(venv) [escritorio@leba fedheonn_ensemble]$ python examples/demo_client.py
      Start training client?: y
Training client...
Elapsed time: 11.42 s
Sending fitted train data...
Data enqueued!: 6f20bbcc-3bbc-4292-9390-11e714b5e84c
Check queue status of partial data? (y/n): y
Successful: 'ENQUEUED'
Check queue status of partial data? (y/n): y
Successful: 'PROCESSING'
Check queue status of partial data? (y/n): n
Receive weights and predict?: y
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.611] seconds
Accuracy on whole TEST data: 91.67
(venv) [escritorio@leba fedheonn_ensemble]$ 

```

Figura 3.52: Finalización de ambos clientes con predicción sobre *test*.

3.7.5. Eficiencia: emisiones y consumo energético

Terminamos este capítulo presentando los distintos experimentos realizados integrando la herramienta *CodeCarbon* directamente en el código de los modelos presentados hasta ahora y en otros equivalentes, para realizar una comparativa en cuanto a consumo energético y eficiencia.

En el primer experimento, correspondiente al fichero *codecarbon_mnist_incremental.py* del directorio *examples*, comparamos el modelo original contra su versión con ensamblaje y contra un tipo de RNA denominada *perceptrón multicapa* (MLPC), realizando el entrenamiento de modo **incremental**; obteniendo el gasto energético, su desglose para cada proceso y las métricas finales resultantes, utilizando el conjunto de datos *MNIST-simp*.

Se divide el conjunto de datos de entrenamiento y se ejecuta el programa con un determinado número de clientes P y clientes por grupo N_g . Para imitar un agregado incremental en el MLPC, simplemente realizamos su *fit* con la sucesiva unión de las submatrices \mathbf{X}_p . Para ser justos con el modelo FedHEONN, durante la primera prueba desactivamos la encriptación para no añadir costes computacionales que el MLPC tampoco considera, y la activamos, por comparación, en la segunda.

En la tabla 3.12 se resumen los hiperparámetros utilizados para estas dos pruebas y en las tablas 3.13 y 3.14 los resultados obtenidos con estas:

MNIST-simp	P	N_g	Encrypt.	λ	F.act.	T	ρ_s	β_s	ρ_f	β_f
FedHEONN	2	1	X	10	logsig	-	-	-	-	-
FedHEONN	2	1	✓	10	logsig	-	-	-	-	-
FedHEONN_ensemble	2	1	X	0.1	logsig	25	0.2	✓	1	X
FedHEONN_ensemble	2	1	✓	0.1	logsig	25	0.2	✓	1	X

MNIST-simp	P	N_g	F.act.	λ	Max.Iter.	Tol.	Solver	Capas ocultas
MLPC	2	1	relu	0.0001	500	0.0001	SGD	(128, 64)

Tabla 3.12: Hiperparámetros ejecutados en `codecarbon_mnist_incremental.py`.

MNIST-simp		Energía cons. (kWh)	Emisiones (CO_2^{eq} kg)	Duración (s)	Exactitud (%)
	FedHEONN	3.5349E-06	7.6857E-07	14.79	91.70
T	fit_group_1	1.5570E-06	3.3853E-07	2.70	-
A	aggregate_partial_group_1	1.0925E-07	2.3753E-08	2.33	-
R	calc_optim_weights_group_1	8.6960E-08	1.8907E-08	2.37	90.92
E	fit_group_2	1.5713E-06	3.4163E-07	2.69	-
A	aggregate_partial_group_2	1.3059E-07	2.8394E-08	2.37	-
S	calc_optim_weights_group_2	7.9835E-08	1.7358E-08	2.33	91.70
	FedHEONN_ensemble	5.4776E-05	1.1910E-05	28.07	93.71
T	fit_group_1	2.2291E-05	4.8467E-06	7.94	-
A	aggregate_partial_group_1	3.1480E-06	6.8445E-07	3.13	-
R	calc_optim_weights_group_1	2.2596E-06	4.9129E-07	2.94	93.53
E	fit_group_2	2.0662E-05	4.4924E-06	7.75	-
A	aggregate_partial_group_2	4.0839E-06	8.8792E-07	3.40	-
S	calc_optim_weights_group_2	2.3316E-06	5.0694E-07	2.91	93.71
	MLPC	1.7256E-04	3.7519E-05	45.32	97.27
T	fit_group_1	7.0460E-05	1.5320E-05	18.22	96.62
.	fit_group_2	1.0210E-04	2.2199E-05	27.10	97.27

Tabla 3.13: Desglose por tareas y modelo del consumo energético, emisiones, duración y exactitud de la primera prueba (**sin encriptación**).

Los ficheros originales se pueden encontrar en la carpeta `emissions_results` del directorio `examples`. Para implementar el MLPC se usó la librería `scikit-learn`.

Como cabría esperar, la energía consumida por el modelo bajo ensamblaje es mayor que el original, y a su vez, el del MLPC es mayor que ambos. El incremento en el consumo de la versión original a la versión con ensamblaje es de **un orden de magnitud mayor**, pasando de unos 3.5×10^{-6} a $5.5 \times 10^{-5} \text{ kWh}$ respectivamente.

Cuando se activa la encriptación, para sendas versiones, se produce un incremento en el consumo energético de **dos ordenes de magnitud**, pasando, aproximadamente, de 3.5×10^{-6} a $2.4 \times 10^{-4} \text{ kWh}$ para el modelo original y de 5.5×10^{-5} a $5.8 \times 10^{-3} \text{ kWh}$ para la versión con ensamblaje (resultados para FedHEONN y FedHEONN_ensemble de las tablas 3.13 y 3.14).

Cabe destacar cómo solo es este último caso, FedHEONN encriptado con ensamblaje, el que consume más energía que el modelo MLPC ensayado ($3.2 \times 10^{-4} \text{ kWh}$), unas 18 veces más. Con todo, en estas pruebas estamos comparados modelos con su conjunto de hiperparámetros ya ajustado, mientras que una prueba más realista incluiría también el tiempo y consumo

MNIST-simp		Energía cons. (kWh)	Emisiones (CO_2^{eq} kg)	Duración (s)	Exactitud (%)
	FedHEONN	2.3780E-04	5.1703E-05	49.37	91.70
T	fit_group_1	5.5734E-06	1.2118E-06	3.00	-
A	aggregate_partial_group_1	2.2911E-07	4.9814E-08	2.33	-
R	calc_optim_weights_group_1	1.1696E-04	2.5430E-05	19.41	90.92
E	fit_group_2	4.3034E-06	9.3565E-07	2.96	-
A	aggregate_partial_group_2	4.1601E-07	9.0449E-08	2.36	-
S	calc_optim_weights_group_2	1.1032E-04	2.3985E-05	19.32	91.70
	FedHEONN_ensemble	5.8543E-03	1.2728E-03	885.20	93.71
T	fit_group_1	1.3924E-04	3.0274E-05	17.74	-
A	aggregate_partial_group_1	6.0752E-06	1.3209E-06	3.22	-
R	calc_optim_weights_group_1	3.1583E-03	6.8668E-04	421.00	93.53
E	fit_group_2	1.1635E-04	2.5296E-05	16.64	-
A	aggregate_partial_group_2	1.2313E-05	2.6770E-06	3.74	-
S	calc_optim_weights_group_2	2.4220E-03	5.2659E-04	422.86	93.71
	MLPC	3.1616E-04	6.8739E-05	45.01	97.27
T	fit_group_1	1.3565E-04	2.9494E-05	19.22	96.62
.	fit_group_2	1.8051E-04	3.9246E-05	25.79	97.27

Tabla 3.14: Desglose por tareas y modelo del consumo energético, emisiones, duración y exactitud de la segunda prueba (**con encriptación**).

energético resultante de dicho ajuste (por ejemplo, mediante *grid search*). Teniendo en cuenta lo anterior, y como el modelo FedHEONN presenta un solo hiperparámetro en su versión original (la *regularización* λ), y otros tres más si hacemos uso del ensamblaje (el número de estimadores base T y el tamaño relativo del parche, ρ_s y ρ_f), cabe esperar que el **consumo energético global de su ajuste sea menor** que para el caso del MLPC u otras RNA, ya que presentan un abanico mucho mayor y más abierto de hiperparámetros y diseños (número de capas, número de neuronas por capa, funciones de activación, regularización, etc.).

Al mismo tiempo también podemos observar, en el desglose de tareas, que, **cuando no se usa la encriptación**, el entrenamiento en los clientes es el proceso que más consume, seguido de la agregación y el cálculo de pesos óptimo, debido a que los dos primeros involucran descomposiciones en valores singulares (SVD), además de otras operaciones, y la última solo unas pocas multiplicaciones de matrices.

En cambio, **cuando se usa la encriptación**, el orden es el inverso, siendo el entrenamiento y agregación las tareas más livianas y la multiplicación de matrices y vectores encriptados, necesarios para el cálculo de pesos óptimos, la más pesada de todas.

En el último experimento, correspondiente al `codecarbon_mnist_experiments.py`, comparamos la **versión original** de FedHEONN, con y sin encriptación (referidos como FedHEONN y FedHEONN_enc), la **versión con ensamblaje**, usando la mejor combinación de $\rho_s \times \rho_f$ vista en la sección 3.7.2, otra alternativa de menor tamaño relativo (denominados FedHEONN_ensemble y FedHEONN_ensemble_alt respectivamente), y el **modelo de MLPC** implementado con *scikit-learn*. Los ejecutamos sobre todos los conjuntos de datos de tipo MNIST presentados en la tabla 3.8; a saber, *MNIST-orig*, *MNIST-simp* y *MNIST-test*.

De nuevo, la configuración de los hiperparámetros utilizada se resume en la tabla 3.15 y los resultados obtenidos en la tabla 3.16:

	P	F.act.	λ	Max.Iter.	Tol.	Solver	Capas ocultas
MLPC	1	<i>relu</i>	0.0001	500	0.0001	SGD	(128, 64)

	P	Encrypt.	F.act.	λ	T	ρ_s	β_s	ρ_f	β_f
FedHEONN	1	X	<i>logsig</i>	10	-	-	-	-	-
FedHEONN_enc	1	✓	<i>logsig</i>	10	-	-	-	-	-
FedHEONN_ensemble	1	X	<i>logsig</i>	0.01	75	0.2	✓	0.8	X
FedHEONN_ensemble_alt	1	X	<i>logsig</i>	0.1	25	0.2	✓	1	X

Tabla 3.15: Hiperparámetros de los modelos en `codecarbon_mnist_experiments.py`.

MNIST-test	Energía cons. (kWh)	Emisiones (CO_2^{eq} kg)	Duración (s)	Precisión (%)
FedHEONN	1.9533E-06	4.2468E-07	2.68	90.74
FedHEONN_enc	3.4259E-04	7.4486E-05	13.84	90.74
FedHEONN_ensemble	1.4024E-04	3.0491E-05	13.75	95.19
FedHEONN_ensemble_alt	7.8264E-05	1.7016E-05	8.52	94.26
MLPC	1.2083E-04	2.6270E-05	13.38	97.41
MNIST-simp				
FedHEONN	2.4421E-06	5.3097E-07	2.86	91.70
FedHEONN_enc	1.4766E-04	3.2104E-05	17.01	91.70
FedHEONN_ensemble	2.1581E-04	4.6922E-05	25.87	93.53
FedHEONN_ensemble_alt	1.1547E-04	2.5106E-05	12.60	93.42
MLPC	2.5489E-04	5.5420E-05	26.81	97.27
MNIST-orig				
FedHEONN	1.6206E-03	3.5236E-04	132.37	81.30
FedHEONN_enc	3.4126E-03	7.4198E-04	352.99	81.32
MLPC	2.3676E-03	5.1476E-04	319.27	97.14

Tabla 3.16: Consumo energético, emisiones, duración y exactitud del experimento `codecarbon_mnist_experiments.py`.

De estos últimos resultados destacamos las siguientes observaciones:

- Tomando el modelo **FedHEONN sin encriptar** como base, el uso de la encriptación supone un coste computacional y, por tanto, energético, muy considerable, con un incremento de alrededor del **112 %** en el mejor de los casos, pasando de 1.6×10^{-6} a 3.4×10^{-3} kWh, en el escenario del dataset más grande, *MNIST-orig*, hasta un **17795 %** para el más pequeño, *MNIST-test*, pasando de 1.9×10^{-6} a 3.4×10^{-4} kWh.
- La gran diferencia en el impacto energético de uno y otro caso, debido al uso o no de la encriptación, nos sugiere que **existe un bagaje considerable** por culpa de las operaciones realizadas con los vectores criptográficos y requeridas por el modelo; tanto que para datasets pequeños, suponen costes computacionales incluso mayores a los inherentes al modelo FedHEONN sin encriptación.

- Por otro lado, el uso del **ensamblaje** también supone, como es evidente, un mayor consumo, al ser necesario entrenar múltiples estimadores base T ; con todo, permite un mayor equilibrio entre eficacia y eficiencia, mediante el debido ajuste de sus parámetros.
- Por ejemplo, el incremento de hasta más de un **4 %** en la exactitud bajo *MNIST-test* con la mejor versión de ensamblaje, **FedHEON_ensemble**, consume un **7500 %** más ($1.4 \times 10^{-4} \text{ kWh}$) que la versión original ($1.9 \times 10^{-6} \text{ kWh}$), pero la versión alternativa, **FedHEON_ensemble_alt**, de menor número de estimadores base T , obtiene una eficacia similar con *solo* el **4000 %** más de consumo ($7.8 \times 10^{-5} \text{ kWh}$). Aunque ambos son incrementos sustanciales respecto a la versión original de referencia, la versión alternativa consume aproximadamente la mitad que la otra, siendo una opción mucho más **eficiente**.

Por último, el modelo MLPC destaca por su alta exactitud y bajo consumo frente a las versiones encriptada y con ensamblaje pero sin encriptar de nuestro modelo ejecutadas para estas pruebas.

Siguen siendo, con todo, aún más costosas que la versión original de FedHEONN sin encriptación, aunque este obtenga métricas bastante peores. Sin embargo, la comparación no es del todo justa, ya que la versatilidad de uno y otro tipo de modelo también es distinta; y mientras **FedHEONN** permite un entrenamiento distribuido, incremental, federado e incluso bajo clientes con conjuntos de datos no IID, ninguna de estas capacidades aplican al otro modelo (MLPC), ni tampoco permite preservar la privacidad de los datos - mediante encriptación u otro tipo de técnica - si no que necesita centralizarlos.

3.8. Conclusiones y futuros trabajos

Por último se trazarán escuetamente las conclusiones del proyecto y posibles líneas de trabajo.

3.8.1. Conclusiones

Demostrada la viabilidad del algoritmo presentado bajo escenarios realistas de entornos federados, las mejoras planteadas en este trabajo han sido implementadas obteniendo resultados mayoritariamente satisfactorios, cuyo resumen listamos a continuación:

- Se ha probado y mejorado la **naturaleza incremental** del algoritmo FedHEONN mediante su desglose en tareas de agregación parcial y cálculo de pesos óptimos en lotes, obteniendo los mismos resultados. Dicho desglose es útil, puesto que la agregación de la información recopilada es un proceso mucho menos costoso que el cálculo de los pesos finales, y se trata de una configuración más realista en un entorno federado en donde no todos los clientes pueden estar disponibles a la vez.
- Se ha incrementado la **capacidad de representación** del modelo mediante la incorporación de la técnica de *ensemble* conocida como Random Patches, obteniendo mejoras notables en las métricas de reconocidos conjuntos de datos usados como *benchmarks*.
- En un intento de **reducción de los tiempos globales** de entrenamiento con ensamblaje, se ha llevado a cabo la parallelización de las tareas de entrenamiento (*cliente*) y agregación parcial y cálculo de pesos óptimos (*coordinador*), con y sin encriptación, resultando en procesos más rápidos bajo determinados escenarios.
- El *sobrecoste* añadido por el manejo y creación de procesos, así como la serialización y reconstrucción de vectores encriptados hicieron que las anteriores tareas solo compensen su parallelización para un **alto número de estimadores base T** .
- El *sobrecoste* de la serialización y reconstrucción de vectores criptográficos es un factor importante y un coste computacional añadido a tener en cuenta también para el prototipo de entorno federado desarrollado, al que se une el hecho de que la librería de encriptación homomórfica elegida no soporte actualmente dispositivos embebidos bajo arquitectura ARM, frustrando alguno de los experimentos de este trabajo.
- El análisis de los consumos energéticos del modelo original con el ensamblaje y otros semejantes nos permitió: **cuantificar y ejemplificar** muchos de los aspectos tratados en el apartado de IA verde/roja (sección 3.3.4), en cuanto a eficacia y eficiencia; y, por otro, **desglosar el consumo de las tareas** involucradas en el proceso federado de FedHEONN, así como el uso – o no – de la encriptación durante el mismo.

3.8.2. Futuros trabajos

En cuanto a futuros trabajos de investigación a los que da pie el modelo presentado, podemos destacar:

- Hacer uso de alguna otra librería de encriptación homomórfica, como *Pyfhel*, para **comparar rendimientos** en las operaciones con vectores y multiplicación de matrices encriptados, así como con la serialización y reconstrucción de los mismos y de sus contextos asociados. Comprobar que **soperten dispositivos embebidos** con arquitectura ARM, como las Raspberry mencionadas en este trabajo, o esperar a que TenSEAL lo haga.
- Integrar la librería de *Google Jax* en el proyecto con el objeto de **acelerar las descomposiciones en valores singulares** (SVD), que son vitales en la agregación de información (*coordinador*) y entrenamiento (*clientes*), ejecutándolos en entornos con TPU's, como pueden ser *Google Cloud TPU* para la coordinación y Raspberry's con coprocesadores **Coral** como clientes.
- Realizar una **comparativa con otros modelos federados** de uso actual, para ver si FedHEONN es más eficiente solo en términos de rondas de comunicación de información necesarias para su entrenamiento o si también en el propio consumo energético requerido por el mismo.
- Acoplar este modelo a otros algoritmos federados de características semejantes para producir una combinación más potente y con una **capacidad de representación más compleja**.
- La elaboración de una **técnica inteligente específica para el ajuste de los hiperparámetros utilizados en Random Patches**. Probando diferentes extremos de los posibles tamaños relativos de los parches, $\rho_s \times \rho_f \in (0, 1)$, se puede ir precisando y encontrando los valores que generan mejores métricas finales de un modo más óptimo que realizando una bruta búsqueda en cuadricula.

3.9. Referencias

En este capítulo disponemos las referencias de los distintos recursos empleados en este Trabajo Final de Grado.

3.9.1. Bibliografía

A continuación listamos los libros, revistas, artículos científicos, páginas web u otro tipo de publicaciones considerados de interés y citados en la memoria:

- [1] Iván Abad. Notacion Big O. <https://www.netmentor.es/entrada/notacion-big-o>, 2021. Accessed: 2024-09-02.
- [2] Jordan Aljbour, Tom Wilson, and Poorvi Patel. Powering Intelligence: Analyzing Artificial Intelligence and Data Center Energy Consumption. <https://www.epri.com/research/products/000000003002028905>, 2023. Accessed: 2024-09-03.
- [3] Dario Amodei and Danny Hernandez. AI and compute. <https://openai.com/research/ai-and-compute>, 2018. Accessed: 2024-09-02.
- [4] Juan Ignacio Barros Arce. La matriz de confusión y sus métricas. <https://www.juanbarrios.com/la-matriz-de-confusion-y-sus-metricas>, 2022. Accessed: 2024-09-02.
- [5] Ayoub Benaissa, Bilal Retiat, Bogdan Cebere, and Alaa Eddine Belfedhal. TenSEAL: A library for encrypted tensor operations using homomorphic encryption. *CoRR*, abs/2104.03152, 2021.
- [6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.
- [7] Benoit Courty, Victor Schmidt, Sasha Luccioni, Goyal-Kamal, MarionCoutarel, Boris Feld, Jérémie Lecourt, LiamConnell, Amine Saboni, Inimaz, supatomic, Mathilde Léval, Luis Blanche, Alexis Cruveiller, ouminasara, Franklin Zhao, Aditya Joshi, Alexis Bogroff, Hugues de Lavoreille, Niko Laskaris, Edoardo Abati, Douglas Blank, Ziyao Wang, Armin Catovic, Marc Alencon, Michal Stechly, Christian Bauer, Lucas Otávio N. de Araújo, JPW, and MinervaBooks. mlco2/CodeCarbon: v2.4.1, May 2024.
- [8] Ministerio de Asuntos Económicos y Transformación Digital. *Programa Nacional de Algoritmos Verdes*. Gobierno de España, 2022.
- [9] Gobierno de España. Ministerio de Asuntos Económicos y Transformación Digital. <https://www.mineco.gob.es>. Accessed: 2023-04-21.
- [10] Gobierno de España. Estrategia Nacional de Inteligencia Artificial 2024 (ENIA). https://portal.mineco.gob.es/es-es/digitalizacionIA/Documents/Estrategia_IA_2024.pdf, 2024. Accessed: 2024-09-03.

- [11] Shereen Mohamed Fawaz, Nahla Belal, Adel ElRefaey, and Mohamed Waleed Fakhr. A comparative study of homomorphic encryption schemes using microsoft SEAL. *Journal of Physics: Conference Series*, 2128(1):012021, Dec 2021.
- [12] Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, Elena Hernández-Pereira, and Beatriz Pérez-Sánchez. FedHEONN: Federated and homomorphically encrypted learning method for one-layer neural networks. *Future Generation Computer Systems*, 149:200–211, 2023.
- [13] Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, and Beatriz Pérez-Sánchez. LANN-SVD: A non-iterative SVD-based learning algorithm for one-layer neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 29(8):3900–3905, 2018.
- [14] Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, Beatriz Pérez-Sánchez, David Martínez-Rego, and Diego Rego-Fernández. A fast learning algorithm for high dimensional problems: an application to microarrays. *ESANN 2016 - 24th European Symposium on Artificial Neural Networks*, pages 283–288, 2016.
- [15] Rainer Gewalt. Supervised vs unsupervised vs reinforcement learning - the fundamental differences. <http://starship-knowledge.com/supervised-vs-unsupervised-vs-reinforcement>, 2022. Accessed: 2023-04-16.
- [16] M. A. Iwen and B. W. Ong. A Distributed and Incremental SVD Algorithm for Agglomerative Data Analysis on Large Networks. *SIAM Journal on Matrix Analysis and Applications*, 37(4):1699–1718, 2016.
- [17] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Benni, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D’Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konecný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrede Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and Open Problems in Federated Learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.
- [18] Dawid Kopczyk. Hyperparameter optimization: Explanation of automatized algorithms. <https://dkopczyk.quantee.co.uk/hyperparameter-optimization>, 2019. Accessed: 2024-06-08.
- [19] Loïc Lannelongue, Jason Grealey, and Michael Inouye. Green Algorithms: Quantifying the carbon emissions of computation. *CoRR*, abs/2007.07610, 2020.
- [20] Peng Li, Jianyi Yang, Mohammad Atiqul Islam, and Shaolei Ren. Making AI Less “Thirsty”: Uncovering and Addressing the Secret Water Footprint of AI Models. *ArXiv*, abs/2304.03271, 2023.

- [21] Koepp Lindsey. Regression (data science part 6) linear regression with math. <https://morioh.com/p/f156f910016e>, 2020. Accessed: 2024-09-02.
- [22] MingYu (Ethen) Liu. K-fold cross validation, grid/random search from scratch. http://ethen8181.github.io/machine-learning/model_selection/model_selection.html, 2018. Accessed: 2024-09-02.
- [23] Gilles Louppe and Pierre Geurts. Ensembles on Random Patches. In Peter A. Flach, Tijl De Bie, and Nello Cristianini, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 346–361, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [24] Abambres M and Ferreira A. Application of ANN in pavement engineering: State-of-art. *University of Coimbra*, 07 2020.
- [25] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 20–22 Apr 2017.
- [26] Shanthababu Pandian. Understand Machine Learning and Its End-to-End Process. <https://shanthababu.com/2020/12/30/understand-machine-learning-and-its-end-to-end-process-2>, 2020. Accessed: 2024-09-02.
- [27] European Parliament, Directorate-General for Internal Policies of the Union, A Herold, P Gailhofer, C Urrutia, S Braungardt, A Köhler, C Scherf, and J Schemmel. *The role of artificial intelligence in the European Green Deal*. European Parliament, 2021.
- [28] European Parliament, Directorate-General for Internal Policies of the Union, A Herold, P Gailhofer, C Urrutia, S Braungardt, A Köhler, C Scherf, and J Schemmel. *The role of artificial intelligence in the European Green Deal*. European Parliament, 2021.
- [29] Ravish Raj. Machine learning glossary: Commonly used terms in machine learning. <https://www.enjoyalgorithms.com/blogs/machine-learning-glossary>, 2021. Accessed: 2024-09-02.
- [30] White House Report. Consumer Data Privacy in a Networked World: A Framework for Protecting Privacy and Promoting Innovation in the Global Digital Economy. *Journal of Privacy and Confidentiality*, 4(2), 2013.
- [31] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *CoRR*, abs/1907.10597, 2019.
- [32] Seofilo. Qué es la ley de moore? <http://seofilo.com/que-es-la-ley-de-moore>, 2016. Accessed: 2024-03-05.
- [33] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. *CoRR*, abs/1906.02243, 2019.

- [34] Qiang Yang, Yang Liu, Yong Cheng, Yan Kang, Tianjian Chen, and Han Yu. *Federated Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer Cham, 2019.

3.9.2. Otras referencias

En este apartado reflejamos todos aquellos manuales que hemos consultado durante la configuración, uso e implementación, tanto de los distintos dispositivos físicos empleados (*hardware*), como durante el desarrollo de los métodos y técnicas descritas para modelo (*software*):

- **Raspberry Pi 4 Product Brief**, PDF disponible en <https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf>
- **Raspberry Pi Official Getting Started Guide**, consulta disponible en <https://www.raspberrypi.com/documentation/computers/getting-started.html>
- **Raspberry Pi Official Documentation**, consulta disponible en <https://www.raspberrypi.com/documentation>
- **Jetson Nano Data Sheet Specification**, PDF disponible en <https://developer.nvidia.com/downloads/embedded/dlc/jetson-nano-system-module-datasheet>
- **Jetson Nano Developer Kit User Guide**, PDF https://developer.download.nvidia.com/embedded/L4T/r32-3-1_Release_v1.0/Jetson_Nano_Developer_Kit_User_Guide.pdf
- **Jetson Nano Getting Started Guide**, consulta disponible en <https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit>
- **The Python Tutorial**, consulta disponible en <https://docs.python.org/3.10/tutorial/index.html>
- **Python Official Documentation**, consulta disponible en <https://docs.python.org/3.10/>
- **Programming Notes**, consulta disponible en <https://www3.ntu.edu.sg/home/ehchua/programming/index.html>, apartado *Server-Side → Python*.
- **Scikit-Learn User Guide**, consulta disponible en https://scikit-learn.org/stable/user_guide.html
- **Scikit-Learn API**, consulta disponible en <https://scikit-learn.org/stable/modules/classes.html>

Título Escalabilidad y eficiencia en Inteligencia Artificial
a través de algoritmos federados verdes

Anexos

Peticionario Escuela Politécnica de Ingeniería de Ferrol
Rúa Mendizábal, s/n, Campus de Esteiro,
15403, Ferrol

Fecha Septiembre de 2024

Autor El Alumno

Fdo. Abel Pampín Rodríguez

Este documento de Anexos está formado por la documentación de partida de este Trabajo Final de Grado, que alberga la propuesta original y las referencias a los artículos en los que se desarrolló el modelo de partida de este proyecto.

Por último se incluyen enlaces al repositorio público que aloja los códigos de programación elaborados, un esquema de su estructura, la configuración del entorno detallada para los distintos dispositivos embebidos empleados en este proyecto y algunas figuras y tablas resultado de los programas mencionados en el capítulo 3.7 de *Resultados finales*.

4.1. Documentación de partida

Artículos científicos

- Enrique Castillo, Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas and Amparo Alonso-Betanzos. *A global optimum approach for One-Layer Neural Networks*. 2002.
- Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, Beatriz Pérez-Sánchez and Amparo Alonso-Betanzos. *A new convex objective function for the supervised learning of single-layer neural networks*. 2009.
- Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, Beatriz Pérez-Sánchez, David Martínez-Rego and Diego Rego-Fernández. *A fast learning algorithm for high dimensional problems: an application to microarrays*. 2016.
- Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas and Beatriz Pérez-Sánchez. *LANN-SVD: A Non-Iterative SVD-Based Learning Algorithm for One-Layer Neural Networks*. 2018.
- Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas and Beatriz Pérez-Sánchez. *Regularized One-Layer Neural Networks for Distributed and Incremental Environments*. 2020.
- Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, Beatriz Pérez-Sánchez and elena Hernández-Pereira. *An effective and efficient green federated learning method for one-layer neural networks*. 2022.
- Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, Beatriz Pérez-Sánchez and elena Hernández-Pereira. *FedHEONN: Federated and homomorphically encrypted learning method for one-layer neural networks*. 2023.

Propuesta

Copia de la propuesta inicial de asignación del Trabajo Final.

Número de traballo 2223_GETI_23
Titulación Grao en Enxeñaría en Tecnoloxías Industriais
Título do proxecto (Título en Galego) Escalabilidade e eficiencia en Intelixencia Artificial a través de algoritmos federados verdes
Título del proyecto (Título en Castelán) Escalabilidad y eficiencia en Inteligencia Artificial a través de algoritmos federados verdes
Project Title (Título en Inglés) Scalability and efficiency in Artificial Intelligence through federated green algorithms
Tipoloxía do proxecto Traballos de investigación: relacionados coa investigación, desenvolvemento e innovación en produtos, procesos e métodos, de carácter teórico, computacional e/ou experimental, que constitúan unha achega á técnica
Grado de dificultade Alta
¿É unha proposta consensuada con un alumno para a súa asignación? Si
Nome do Titor/a Óscar Fontenla Romero
Nome do Titor/a (Só se hai dous titores)
Empresa do Titor (No caso de non ser da UDC)
Antecedentes detallados do proxecto <p>No hay duda que, a día de hoy, la Inteligencia Artificial (IA), el Aprendizaje Automático (ML) y sus utilidades están tomando cada día mayor relevancia en todos los aspectos de nuestras vidas. Dentro del ML existe una tendencia relativamente reciente, denominada como Aprendizaje Federado (FL), que se está convirtiendo en una de las líneas de investigación más activas en este campo. Debido precisamente al gran abanico de aplicaciones que se pueden beneficiar del uso de algún modelo de aprendizaje automático, y al cada vez mayor conjunto de datos con el que trabajan, el actual entrenamiento centralizado habitual en estos presenta unos límites - tanto a nivel de infraestructura, como de consumo y privacidad - que ya no pueden ser ignorados. El FL supone un nuevo enfoque a los anteriores obstáculos, combinando los beneficios de modelos que trabajan sobre datos distribuidos junto con una naturaleza privativa por diseño, en donde no existe una comunicación en crudo de los datos locales, sino que cada nodo realiza paralelamente entrenamientos parciales sobre sus propios datos para luego enviar estos resultados a un agregador o coordinador central para obtener así el modelo global, que será compartido por todos. El método de agregación más usado y conocido en estos escenarios es el denominado FedAvg, que, tras seleccionar una colección aleatoria de nodos cliente - a los que ordena entrenar localmente su modelo -, recibe de ellos las actualizaciones de sus pesos, que promedia según una serie de criterios y devuelve, por último, los resultados a todos los nodos, poniendo fin así a un ciclo de entrenamiento. Este proceso se repite varias veces hasta que el modelo global alcance un desempeño satisfactorio. Este tipo de algoritmos se convierten así en candidatos óptimos para ir de la mano con el auge del Internet of Things (IoT) y la llamada Industria 4.0, que representan la convergencia creciente de las tecnologías de la información y las tecnologías operativas propias de procesos industriales. De cara al IoT, la mayoría de estos dispositivos tienen capacidades limitadas, y al utilizar aprendizaje federado pueden colaborar para entrenar modelos más complejos aún con pocos recursos. De cara a la industria, el aprendizaje federado ofrece privacidad y seguridad en la gestión de datos críticos en estos entornos, lo que es clave no solo para su adopción, sino que sirve también de catalizador para nuevas sinergias, permitiendo la colaboración de diferentes entidades con informaciones sensibles para el entrenamiento de un modelo conjunto que beneficie a ambas, por ejemplo.</p>

Obxeto detallado do proxecto

En este Trabajo Final de Grado se abordará el desarrollo de técnicas de aprendizaje federado con una perspectiva “verde” (green IA), en sintonía con la manifestación de interés del Programa Nacional de Algoritmos Verdes publicado por el Ministerio de Asuntos Económicos y Transformación Digital [1]. El propósito es el de fomentar el desarrollo de una inteligencia artificial (IA) verde por diseño (Green by design) y medioambientalmente sostenible, ya que la inteligencia artificial por si misma plantea actualmente una serie de amenazas medioambientales. El uso de tecnologías de la información y de las comunicaciones supone hoy en día entre un 5% y un 9% del consumo total de electricidad en todo el mundo, y podría llegar al 20% en 2030, según el informe “The Role of Artificial Intelligence in the European Green Deal” realizado por el Parlamento Europeo. Esto se traduce en unas emisiones de gases de efecto invernadero de entre 1,1 y 1,3 Gt de CO₂ en 2020. Los recursos de computación necesarios para entrenar modelos de inteligencia artificial se están doblando cada 3-4 meses desde el año 2012 a medida que se buscan modelos más precisos. En esta línea de investigación se sitúa el último [2] de una serie de trabajos desarrollados en la UDC, donde se explora la aplicación de un nuevo método de entrenamiento regularizado para redes neuronales de una sola capa (que presenta una serie de ventajas particulares - solución no-iterativa, convergencia en escenarios heterogéneos etc.-) en un entorno federado. El trabajo del alumno, que tendrá de base el citado algoritmo, ampliará y experimentará posibles nuevas mejoras a este, como su uso junto con métodos de ensemble o su participación como pieza de modelos más complejos. Los algoritmos serán implementados para que puedan ser ejecutados en dispositivos embebido (Raspberry PI, Jetson Nano, etc.), que son de gran interés en entornos industriales y de IoT.[1] Ministerio de Asuntos Económicos y Transformación Digital, “PNAV: Programa Nacional de Algoritmos Verdes”, 2022.

https://portal.mineco.gob.es/RecursosNoticia/mineco/prensa/noticias/2022/20221213_plan_algoritmos_verdes.pdf[2] Óscar Fontenla-Romero, Bertha Guijarro-Berdiñas, Elena Hernández-Pereira, Beatriz Pérez-Sánchez, "An effective and efficient green federated learning method for one-layer neural networks", 2022.

Alcance detallado do proxecto

- Análisis y familiarización con los algoritmos más idóneos para abordar el problema.- Estudio del estado del arte del aprendizaje federado.- Estudio y familiarización de los dispositivos embebidos.- Desarrollo e implementación en Python de los algoritmos de aprendizaje automático para el entorno federado.- Diseño y desarrollo de un estudio experimental con conjuntos de datos de clasificación o predicción.- Análisis de los resultados del estudio experimental en términos de eficacia, eficiencia y consumo eléctrico.- Desarrollo de la documentación del TFG.

4.2. Código de programación

Todo el código de programación elaborado para la realización de las distintas pruebas de este Trabajo Final de Grado se encuentra hospedado bajo el servicio gratuito **GitHub**, que ofrece un entorno en la nube para el alojamiento de código fuente y su versionado mediante la herramienta *Git*.

Repositorio y estructura

Se puede acceder al repositorio a través del siguiente enlace https://github.com/abeludc93/fedheonn_ensemble. Su estructura básica y descripción se ilustran en la siguiente página.

A mayores, cabe destacar que el repositorio dispone de dos ramas o *branches* distintas, una denominada **master** y otra llamada **codecarbon**. En la primera se deja una versión simplificada del algoritmo, con menos ejemplos pero sólo con las dependencias estrictamente requeridas por el programa base. En la otra se encuentran, a mayores, los experimentos y dependencias necesarias para el estudio medioambiental y energético realizado junto con la librería CodeCarbon.

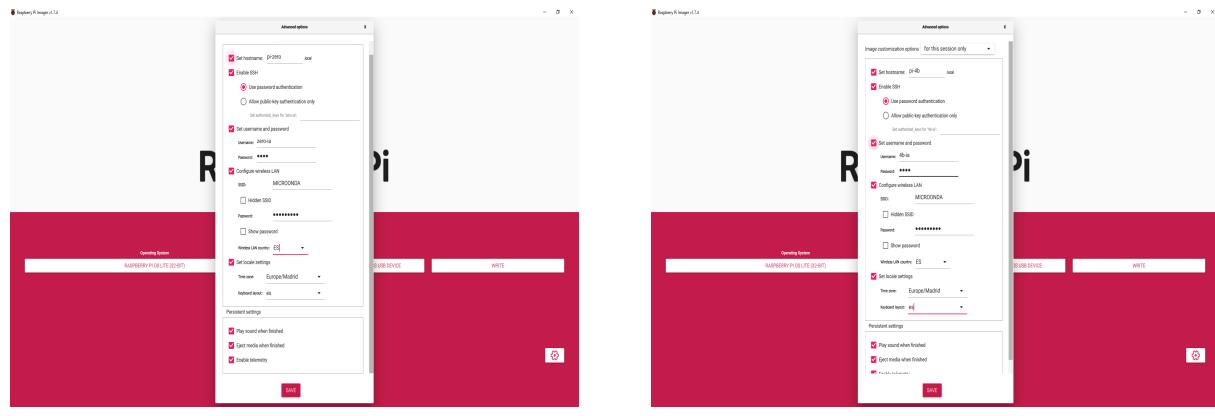
Instalación y ejemplos

Dentro del repositorio existe un fichero denominado *requirements.txt* que lista las dependencias (librerías externas) necesarias para la ejecución del programa, además de la versión empleada; y, por último, se encuentra otro fichero de texto llamado *README.md*, donde se dan unas instrucciones para la correcta instalación de estas en un equipo o entorno con Python.

```
fedheon-ensemble
├── algorithm: paquetería con la implementación de algoritmos FedHEONN
│   ├── fedHEONN-clients
│   ├── fedHEONN-coordinators
│   ├── activation-functions
│   └── metrics
├── api: paquetería con el servidor REST API y cliente para su consumo
│   ├── client
│   ├── server
│   └── utils
├── auxiliary: módulos para carga de configuraciones y logging
│   ├── decorators
│   ├── config-loader
│   └── logger
├── datasets: directorio con algunas de las bases de datos empleadas
│   ├── DryBean-Dataset
│   ├── MiniBooNE-Dataset
│   └── ...
├── examples: paquetería con todos los ejemplos y experimentos realizados
│   ├── utils
│   ├── mnist
│   ├── demo-server
│   ├── demo-client
│   └── ...
├── README
└── requirements
└── demo-config
```

4.3. Configuración del entorno

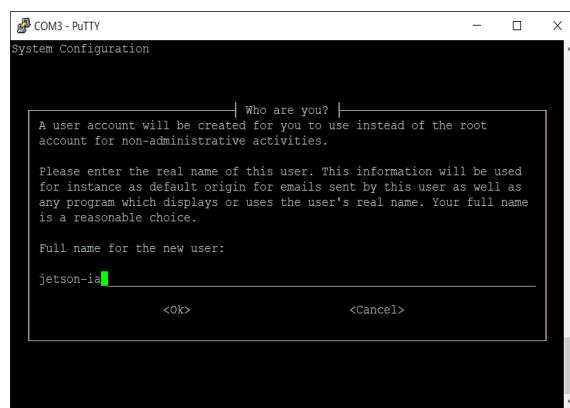
La configuración de los dispositivos embebidos incluyó la instalación de un sistema operativo sin interfaz gráfica (llamado *headless-config*). Para la Raspberry Pi, se usó Raspberry Pi OS Lite, configurando la red y el acceso SSH antes de conectarse mediante terminal para actualizar e instalar Python y las librerías necesarias. En el caso de Jetson Nano, se realizó un proceso similar, pero con conexión inicial por microUSB, seguida de configuración de red vía Ethernet, al no disponer de tarjeta inalámbrica. Ambos procesos se basaron en sendas guías oficiales de referencia que enlazamos en la sección 3.9.2.



(a) Menú de configuración de Raspberry Pi Imager Installer para modelo Pi Zero

(b) Menú de configuración de Raspberry Pi Imager Installer para modelo Pi 4B

Figura 4.1: ① - Configuración e instalación de SO en dispositivos Raspberry

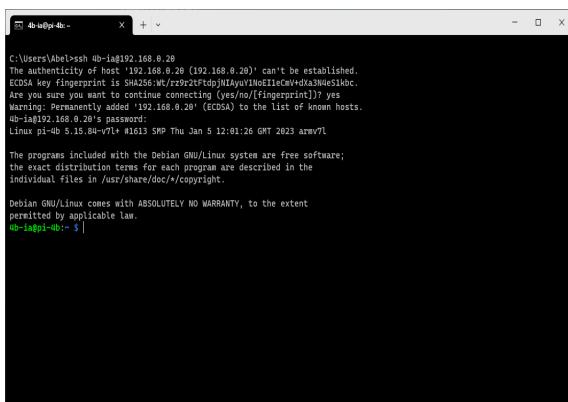


(a) Creando usuario

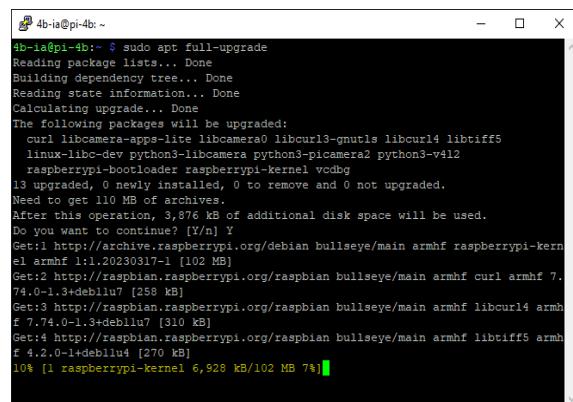


(b) Instalación terminada, primer inicio

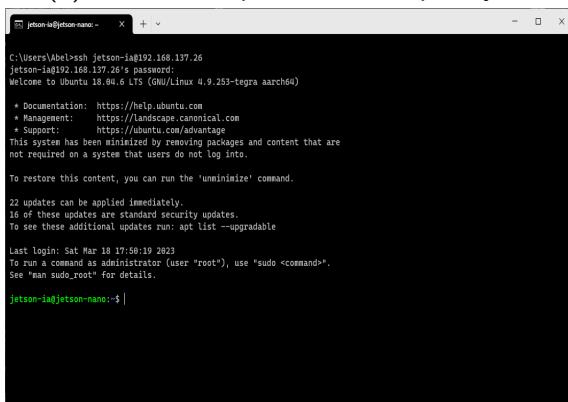
Figura 4.2: ② - Configuración e instalación de SO para Jetson Nano



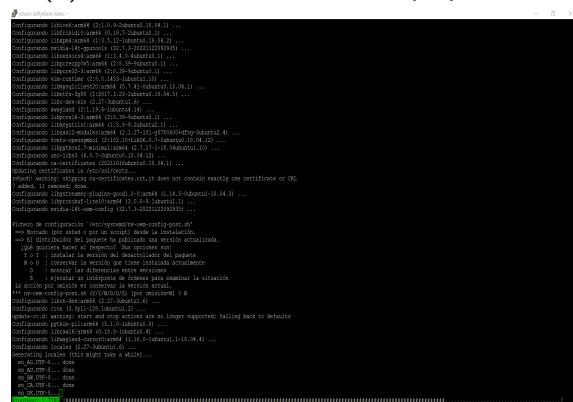
(a) Accediendo por SSH a Raspberry Pi



(b) Actualizando e instalando paqueterías



(c) Accediendo mediante SSH al dispositivo



(d) Instalando dependencias necesarias

Figura 4.3: ③ - Conexión mediante SSH y actualización e instalación de programas y librerías

En las figuras 4.1 y 4.2 podemos ver parte del proceso de instalación del SO en los dispositivos embebidos, y la posterior instalación y actualización de librerías y lenguajes de programación en 4.3.

4.4. Tablas y figuras de resultados

```

INFO    - Cross validation split: 5
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [4.412] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.070] seconds
INFO    - Cross validation split: 6
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [4.430] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.070] seconds
INFO    - Cross validation split: 7
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [4.446] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.070] seconds
INFO    - Cross validation split: 8
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [4.430] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.069] seconds
INFO    - Cross validation split: 9
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [4.437] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.069] seconds
INFO    - Cross validation split: 10
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [4.432] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.070] seconds
INFO    - WITHOUT BAGGING:
   LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
0  0.01      None     None     None     None     88.14857143 3.13787799
1  0.10      None     None     None     None     87.98857143 3.22183505
INFO    - WITH BAGGING:
   LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
181 0.01        75    True    False    0.2    0.8  95.94476190 2.36870509
261 0.01        100   True    False    0.2    0.8  95.94476190 2.14874826
243 0.01        100   True    True     0.2    1.0  95.94349206 2.11871113
581 0.10        100   True    False    0.2    0.8  95.94222222 1.53402252
621 0.10        100   False   False    0.2    0.8  95.94095238 1.96949689
INFO    - Elapsed time running [main]: [71872.350] seconds
C:\Users\Abel\Desktop\FedHEONN\fedheonn_ensemble\examples>

```

Figura 4.4: Salida del programa `mnist_gridsearchcv_remote.py`

```

INFO    - Cross validation split: 5
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [0.011] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.938] seconds
INFO    - Cross validation split: 6
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [0.011] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.942] seconds
INFO    - Cross validation split: 7
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [0.011] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.939] seconds
INFO    - Cross validation split: 8
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [0.011] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.943] seconds
INFO    - Cross validation split: 9
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [0.011] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.937] seconds
INFO    - Cross validation split: 10
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [0.011] seconds
INFO    - Starting timer to [predict]
INFO    - Elapsed time running [predict]: [0.936] seconds
INFO    - WITHOUT BAGGING:
   LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
1  10      None     None     None     None     80.52337948 0.30907814
0  1      None     None     None     None     80.51871575 0.29858924
INFO    - WITH BAGGING:
   LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
210 10        2    False   True     0.1    0.7  84.21466572 0.33665820
211 10        2    False   True     0.1    0.8  84.21466572 0.33665820
212 10        2    False   True     0.1    0.9  84.21466572 0.33665820
215 10        2    False   True     0.2    0.9  83.93076628 0.37849137
213 10        2    False   True     0.2    0.7  83.93076628 0.37849137
INFO    - Elapsed time running [main]: [3496.061] seconds
C:\Users\Abel\Desktop\FedHEONN\fedheonn_ensemble\examples>

```

Figura 4.5: Salida del programa `skin_gridsearchcv_remote.py`

```

INFO - Elapsed time running [aggregate]: [1.036] seconds
INFO - Starting timer to [predict]
INFO - Elapsed time running [predict]: [0.945] seconds
INFO - Cross validation split: 6
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [1.224] seconds
INFO - Starting timer to [predict]
INFO - Elapsed time running [predict]: [1.033] seconds
INFO - Cross validation split: 7
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [1.153] seconds
INFO - Starting timer to [predict]
INFO - Elapsed time running [predict]: [1.062] seconds
INFO - Cross validation split: 8
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [1.312] seconds
INFO - Starting timer to [predict]
INFO - Elapsed time running [predict]: [1.088] seconds
INFO - Cross validation split: 9
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [1.221] seconds
INFO - Starting timer to [predict]
INFO - Elapsed time running [predict]: [1.213] seconds
INFO - Cross validation split: 10
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [1.196] seconds
INFO - Starting timer to [predict]
INFO - Elapsed time running [predict]: [1.224] seconds
INFO - WITHOUT BAGGING:
    LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
0 0.01 None None None None None 90.68937544 1.00000706
1 0.10 None None None None None 90.10166921 0.97254529
2 1.00 None None None None None 88.71622783 0.68291349
3 10.00 None None None None None 87.60354299 0.67143714
INFO - WITH BAGGING:
    LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
979 0.01 10 True False 0.8 1.0 91.01477422 1.02630362
989 0.01 10 True False 0.9 1.0 90.99383195 0.96452057
369 0.01 2 False False 0.7 1.0 90.97276844 0.97906789
949 0.01 10 True False 0.5 1.0 90.93076265 1.02248252
2959 0.01 200 True False 0.6 1.0 90.90988651 0.88862982
INFO - Elapsed time running [main]: [462709.809] seconds

```

C:\Users\Abel\Desktop\FedIEONN\fedheonn_ensemble\examples>python drybean_gridsearchcv_remote.py

Figura 4.6: Salida del programa drybean_gridsearchcv_remote.py

```

INFO - Elapsed time running [aggregate]: [0.533] seconds
INFO - Cross validation split: 2
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.515] seconds
INFO - Cross validation split: 3
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.551] seconds
INFO - Cross validation split: 4
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.471] seconds
INFO - Cross validation split: 5
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.533] seconds
INFO - Cross validation split: 6
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.529] seconds
INFO - Cross validation split: 7
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.519] seconds
INFO - Cross validation split: 8
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.536] seconds
INFO - Cross validation split: 9
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.377] seconds
INFO - Cross validation split: 10
INFO - Starting timer to [aggregate]
INFO - Elapsed time running [aggregate]: [0.528] seconds
INFO - WITHOUT BAGGING:
    LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
0 0.01 None None None None None 0.00007847 0.00008703
1 0.10 None None None None None 0.00007847 0.00008703
2 1.00 None None None None None 0.00007847 0.00008704
3 10.00 None None None None None 0.00007924 0.00008718
INFO - WITH BAGGING:
    LAMBDA N_ESTIMATORS B_SAMPLES B_FEATS P_SAMPLES P_FEATS METRIC_MEAN METRIC_STD
2139 0.01 75 True False 0.4 1.0 0.00007832 0.00008686
2159 0.01 75 True False 0.6 1.0 0.00007832 0.00008685
2329 0.01 75 False False 0.3 1.0 0.00007832 0.00008685
2169 0.01 75 True False 0.7 1.0 0.00007832 0.00008686
2149 0.01 75 True False 0.5 1.0 0.00007832 0.00008685
INFO - Elapsed time running [main]: [52468.914] seconds

```

C:\Users\Abel\Desktop\FedIEONN\fedheonn_ensemble\examples>python carbon_gridsearchcv_remote.py

Figura 4.7: Salida del programa carbon_gridsearchcv_remote.py

MNIST

Fila	λ	T	β_s	β_f	ρ_s	ρ_f	Precisión % (μ, σ)
0	0.01	75	✓	X	0.2	0.8	95.94 2.37
...							
64	0.1	75	✓	X	0.3	0.7	95.23 1.38
...							
128	0.01	75	X	X	0.3	0.9	94.83 2.42
...							
192	0.01	25	✓	X	0.3	1	94.59 2.56
...							
256	0.1	100	✓	X	0.6	0.8	94.27 1.62
...							
320	0.01	25	X	✓	0.2	1	93.96 2.39
...							
384	0.1	100	✓	✓	0.5	1	93.71 1.69
...							
448	0.1	50	X	X	0.5	0.8	93.32 2.13
...							
512	0.1	100	X	✓	0.5	0.9	92.84 1.70
...							
576	0.01	75	X	✓	0.5	0.7	92.36 1.68

Tabla 4.1: Algunos resultados del *grid-search* de `mnist_gridsearchcv_remote.py`**MNIST**

λ	Precisión % (μ, σ)
0.01	88.15 3.14
0.1	87.99 3.22

Tabla 4.2: Métricas para regularizaciones usadas sin *ensemble* en MNIST

SKIN									
Fila	λ	T	β_s	β_f	ρ_s	ρ_f	Precisión % (μ, σ)		
0	10	2	X	✓	0.1	0.7	84.21	0.34	
...									
36	10	2	✓	✓	0.4	0.9	83.59	0.36	
...									
72	10	2	✓	X	0.2	0.9	81.51	0.32	
...									
108	1	2	X	X	0.5	0.8	81.21	0.34	
...									
144	10	10	✓	X	0.3	0.8	80.57	0.34	
...									
180	1	10	X	X	0.3	0.9	80.42	0.34	
...									
216	1	5	X	X	0.4	0.7	80.36	0.33	
...									
252	10	5	✓	✓	0.2	0.7	79.92	0.30	
...									
288	1	5	✓	✓	0.4	0.8	79.85	0.29	
...									
324	1	10	✓	✓	0.2	0.9	79.62	0.28	
...									

Tabla 4.3: Algunos resultados del *grid-search* de skin_gridsearchcv_remote.py

SKIN		
λ	Precisión % (μ, σ)	
10	80.52	0.31
1	80.52	0.30

Tabla 4.4: Métricas para regularizaciones usadas sin *ensemble* en Skin

MINIBOONE

Fila	λ	T	β_s	β_f	ρ_s	ρ_f	Precisión % (μ, σ)
0	0.01	2	X	X	0.05	0.9	86.85 0.39
...							
14	0.01	2	✓	X	0.1	0.7	86.10 0.29
...							
28	0.01	20	X	X	0.1	0.9	85.76 0.31
...							
43	0.01	2	✓	✓	0.15	0.7	85.56 0.30
...							
57	0.01	2	X	✓	0.1	0.9	85.46 0.22
...							
72	0.01	10	✓	✓	0.1	0.7	85.27 0.26
...							
86	0.01	20	✓	✓	0.15	0.9	85.16 0.28
...							
100	0.01	10	X	✓	0.05	0.7	85.02 0.35
...							
115	0.01	5	X	✓	0.1	0.5	84.66 0.32
...							
129	0.01	2	X	✓	0.15	0.5	84.55 0.43

Tabla 4.5: Algunos resultados del *grid-search* de `miniboone_gridsearchcv_remote.py`

MINIBOONE

λ	Precisión % (μ, σ)
0.01	85.34 0.30

Tabla 4.6: Métricas para regularizaciones usadas sin *ensemble* en MiniBoone

DRY BEAN

Fila	λ	T	β_s	β_f	ρ_s	ρ_f	Precisión % (μ, σ)
0	0.01	10	✓	X	0.8	1	91.01 1.03
...							
1280	0.01	75	✓	X	0.1	0.6	89.08 0.87
...							
2560	1	10	X	X	0.3	0.9	88.52 0.69
...							
3840	1	75	X	✓	0.6	0.7	88.10 0.67
...							
5120	0.1	25	✓	✓	0.8	0.5	87.55 0.90
...							
6400	0.1	2	✓	X	0.1	0.8	86.20 0.89
...							
7680	0.01	75	✓	✓	0.7	0.3	82.54 0.70
...							
8960	10	200	X	✓	0.8	0.4	77.45 0.67
...							
10240	1	100	X	✓	0.2	0.3	71.91 0.81
...							
11520	10	2	X	✓	0.4	0.4	58.83 1.14

Tabla 4.7: Algunos resultados del *grid-search* de drybean_gridsearchcv_remote.py**DRY BEAN**

λ	Precisión % (μ, σ)
0.01	90.69 1.01
0.1	90.10 0.97
1	88.72 0.68
10	87.60 0.67

Tabla 4.8: Métricas para regularizaciones usadas sin *ensemble* en Dry Bean

CARBON NANOTUBES

Fila	λ	T	β_s	β_f	ρ_s	ρ_f	MSE (μ, σ)	
0	0.01	75	✓	X	0.4	1	7.8320E-05	8.6860E-05
...								
1280	0.01	5	X	✓	1	1	4.2933E-03	1.4246E-04
...								
2560	0.01	100	X	✓	0.5	1	8.5784E-03	1.6715E-04
...								
3840	0.1	50	X	X	1	0.7	1.2420E-02	1.9559E-04
...								
5120	0.01	25	X	X	0.1	0.6	1.4846E-02	2.6839E-04
...								
6400	0.01	75	✓	✓	0.9	0.6	1.9217E-02	2.8718E-04
...								
7680	0.1	100	✓	✓	0.7	0.4	2.9835E-02	3.7920E-04
...								
8960	0.01	10	✓	✓	0.5	0.2	3.8300E-02	4.6811E-04
...								
10240	0.01	200	✓	X	0.3	0.2	5.5343E-02	7.3884E-04
...								
11520	1	75	✓	✓	0.3	0.1	8.4262E-02	1.2766E-03
...								

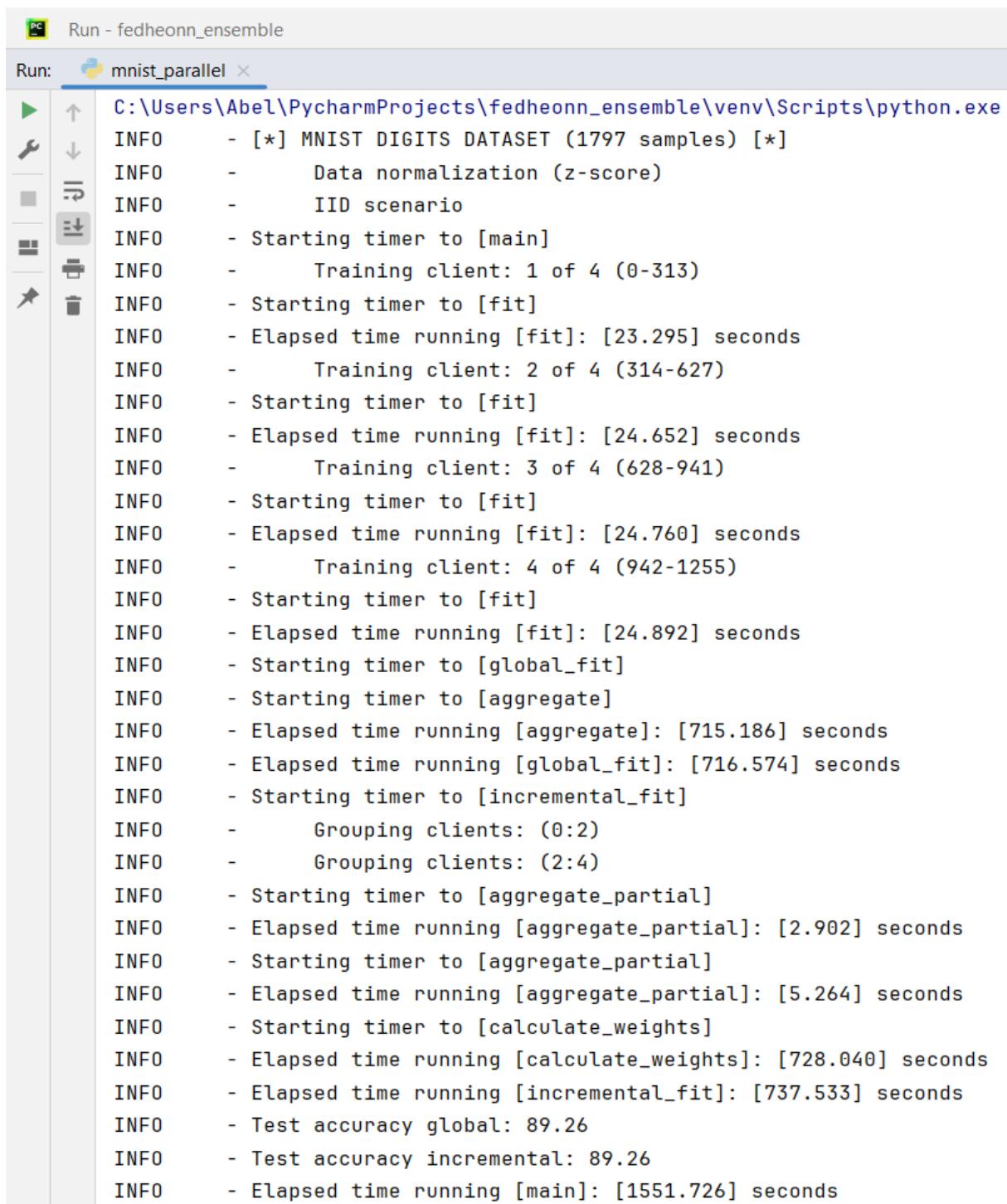
Tabla 4.9: Algunos resultados del *grid-search* de carbon_nanotubes_gridsearchcv_remote.py

CARBON NANOTUBES

λ	MSE (μ, σ)	
0.01	7.8466E-05	8.7030E-05
0.1	7.8466E-05	8.7031E-05
1	7.8474E-05	8.7045E-05
10	7.9242E-05	8.7180E-05

Tabla 4.10: Métricas para regularizaciones usadas sin *ensemble* en Carbon Nanotubes

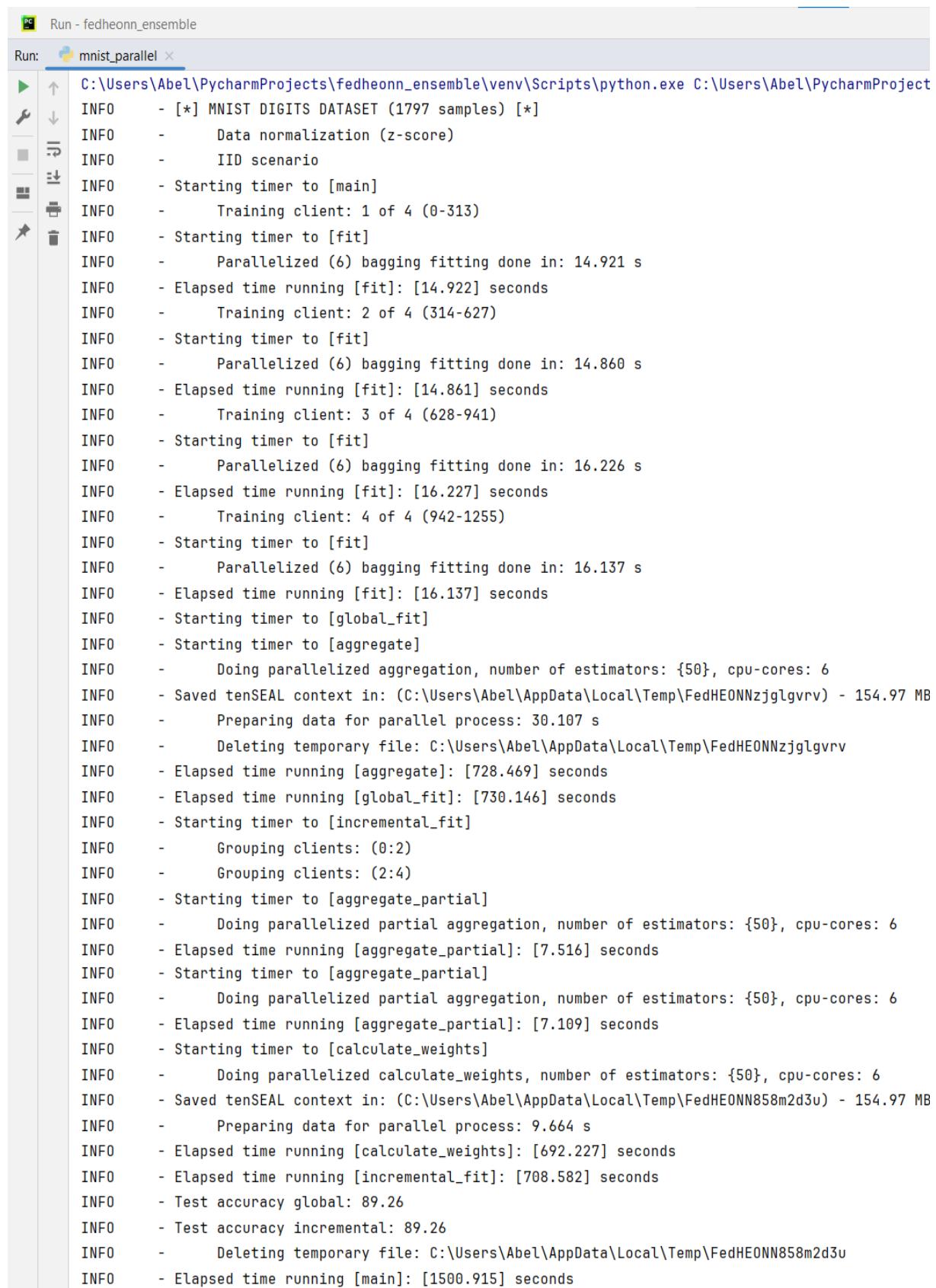
Los resultados están ordenados por métrica de manera ascendente para los casos de clasificación y de manera descendente en los de regresión. Como se puede apreciar en las tablas anteriores, la técnica de ensamblaje solo mejora notablemente la obtenida por el modelo original en los conjuntos *MNIST-test* y *Skin*.



The screenshot shows the PyCharm IDE interface with the terminal window open. The title bar says "Run - fedheonn_ensemble" and the tab bar shows "Run: mnist_parallel". The terminal output displays the execution of a Python script named "mnist_parallel.py". The log level is set to INFO. The output details the process of loading the MNIST digits dataset, normalizing the data, and performing an IID scenario. It tracks the progress of training four clients sequentially, with each client taking approximately 24 seconds. After all clients are trained, it moves on to global fitting, which takes about 716 seconds. Then, it performs incremental fitting, grouping clients into two groups of two, and calculating weights, which takes about 728 seconds. Finally, it calculates the test accuracy, which is 89.26%.

```
C:\Users\Abel\PycharmProjects\fedheonn_ensemble\venv\Scripts\python.exe
INFO    - [*] MNIST DIGITS DATASET (1797 samples) [*]
INFO    - Data normalization (z-score)
INFO    - IID scenario
INFO    - Starting timer to [main]
INFO    - Training client: 1 of 4 (0-313)
INFO    - Starting timer to [fit]
INFO    - Elapsed time running [fit]: [23.295] seconds
INFO    - Training client: 2 of 4 (314-627)
INFO    - Starting timer to [fit]
INFO    - Elapsed time running [fit]: [24.652] seconds
INFO    - Training client: 3 of 4 (628-941)
INFO    - Starting timer to [fit]
INFO    - Elapsed time running [fit]: [24.760] seconds
INFO    - Training client: 4 of 4 (942-1255)
INFO    - Starting timer to [fit]
INFO    - Elapsed time running [fit]: [24.892] seconds
INFO    - Starting timer to [global_fit]
INFO    - Starting timer to [aggregate]
INFO    - Elapsed time running [aggregate]: [715.186] seconds
INFO    - Elapsed time running [global_fit]: [716.574] seconds
INFO    - Starting timer to [incremental_fit]
INFO    - Grouping clients: (0:2)
INFO    - Grouping clients: (2:4)
INFO    - Starting timer to [aggregate_partial]
INFO    - Elapsed time running [aggregate_partial]: [2.902] seconds
INFO    - Starting timer to [aggregate_partial]
INFO    - Elapsed time running [aggregate_partial]: [5.264] seconds
INFO    - Starting timer to [calculate_weights]
INFO    - Elapsed time running [calculate_weights]: [728.040] seconds
INFO    - Elapsed time running [incremental_fit]: [737.533] seconds
INFO    - Test accuracy global: 89.26
INFO    - Test accuracy incremental: 89.26
INFO    - Elapsed time running [main]: [1551.726] seconds
```

Figura 4.8: Salida de `mnist_parallel.py` en serie.



The screenshot shows the PyCharm IDE's run interface. The title bar says "Run - fedheonn_ensemble". The run configuration dropdown is set to "mnist_parallel". The main area displays the command-line output of the "mnist_parallel.py" script. The output shows the process of loading the MNIST digits dataset, normalizing the data, and performing parallelized bagging fitting across four clients. It also shows the preparation of a tenSEAL context, the deletion of temporary files, and the final test accuracy results.

```
C:\Users\Abel\PycharmProjects\fedheonn_ensemble\venv\Scripts\python.exe C:\Users\Abel\PycharmProject
INFO    - [*] MNIST DIGITS DATASET (1797 samples) [*]
INFO    -      Data normalization (z-score)
INFO    -      IID scenario
INFO    - Starting timer to [main]
INFO    -      Training client: 1 of 4 (0-313)
INFO    - Starting timer to [fit]
INFO    -      Parallelized (6) bagging fitting done in: 14.921 s
INFO    -      Elapsed time running [fit]: [14.922] seconds
INFO    -      Training client: 2 of 4 (314-627)
INFO    - Starting timer to [fit]
INFO    -      Parallelized (6) bagging fitting done in: 14.860 s
INFO    -      Elapsed time running [fit]: [14.861] seconds
INFO    -      Training client: 3 of 4 (628-941)
INFO    - Starting timer to [fit]
INFO    -      Parallelized (6) bagging fitting done in: 16.226 s
INFO    -      Elapsed time running [fit]: [16.227] seconds
INFO    -      Training client: 4 of 4 (942-1255)
INFO    - Starting timer to [fit]
INFO    -      Parallelized (6) bagging fitting done in: 16.137 s
INFO    -      Elapsed time running [fit]: [16.137] seconds
INFO    - Starting timer to [global_fit]
INFO    - Starting timer to [aggregate]
INFO    -      Doing parallelized aggregation, number of estimators: {50}, cpu-cores: 6
INFO    -      Saved tenSEAL context in: (C:\Users\Abel\AppData\Local\Temp\FedHEONNzjglgvrv) - 154.97 MB
INFO    -      Preparing data for parallel process: 30.107 s
INFO    -      Deleting temporary file: C:\Users\Abel\AppData\Local\Temp\FedHEONNzjglgvrv
INFO    -      Elapsed time running [aggregate]: [728.469] seconds
INFO    -      Elapsed time running [global_fit]: [730.146] seconds
INFO    - Starting timer to [incremental_fit]
INFO    -      Grouping clients: (0:2)
INFO    -      Grouping clients: (2:4)
INFO    - Starting timer to [aggregate_partial]
INFO    -      Doing parallelized partial aggregation, number of estimators: {50}, cpu-cores: 6
INFO    -      Elapsed time running [aggregate_partial]: [7.516] seconds
INFO    - Starting timer to [aggregate_partial]
INFO    -      Doing parallelized partial aggregation, number of estimators: {50}, cpu-cores: 6
INFO    -      Elapsed time running [aggregate_partial]: [7.109] seconds
INFO    - Starting timer to [calculate_weights]
INFO    -      Doing parallelized calculate_weights, number of estimators: {50}, cpu-cores: 6
INFO    -      Saved tenSEAL context in: (C:\Users\Abel\AppData\Local\Temp\FedHEONN858m2d3u) - 154.97 MB
INFO    -      Preparing data for parallel process: 9.664 s
INFO    -      Elapsed time running [calculate_weights]: [692.227] seconds
INFO    -      Elapsed time running [incremental_fit]: [708.582] seconds
INFO    -      Test accuracy global: 89.26
INFO    -      Test accuracy incremental: 89.26
INFO    -      Deleting temporary file: C:\Users\Abel\AppData\Local\Temp\FedHEONN858m2d3u
INFO    -      Elapsed time running [main]: [1500.915] seconds
```

Figura 4.9: Salida de mnist_parallel.py en paralelo.