UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Software de Topología Computacional para el análisis estructural de redes neuronales subyacentes a modelos de Deep Learning

Realizado por:

José Manuel Ros Rodrigo

Tutelado por:

José Luis Ansorena Barasoain

Julio Rubio García

Logroño, julio, 2023

Resumen/Abstract

Resumen

En este trabajo fin de grado se continúa la labor iniciada con el TFG de Matemáticas titulado «*Homología persistente como herramienta de análisis de redes neuronales*» defendido en julio de 2022, en el que tuvimos la oportunidad de analizar los aspectos matemáticos de la herramienta. Sin embargo, nos faltó alcance para seguir estudiando la herramienta y experimentar con ella, lo que ha motivado la realización del presente trabajo. En concreto, con este proyecto trataremos de mejorar el software producido en el anterior trabajo, así como dar respuesta a la siguiente pregunta: ¿Qué significado tiene la homología persistente en una red neuronal?

En primer lugar, exponemos una breve introducción al problema, seguida de los aspectos relacionados con la planificación del proyecto. En segundo lugar, analizamos el software del anterior trabajo y proponemos un nuevo diseño. Finalmente, desarrollamos e implementamos una serie de soluciones que empleamos sobre una colección de ejemplos reales para extraer conclusiones sobre la herramienta.

Abstract

In this final degree project, we continue the work initiated with the mathematics project entitled «*Persistent homology as a tool to analyze artificial neuronal networks*» presented in July 2022, in that project we had the opportunity of analyzing the mathematical aspects of the tool. However, we lacked scope to further study the tool and experiment with it, which motivated the present work. Specifically, with this project, we will attempt to improve the software produced in the previous work and address the following question: What is the meaning of persistent homology in a neural network?

Firstly, we provide a brief introduction to the problem, followed by aspects related to project planning. Secondly, we analyze the software from the previous work and propose a new design. Finally, we develop and implement a series of solutions that we apply to a collection of real examples to draw conclusions about the tool.

Índice general

Capítulo 1
Introducción

Nuestro objetivo con este trabajo es el de responder a las preguntas de cómo se calcula la homología persistente de una red neuronal de manera computacional y, más importante aún, qué significado tiene la homología persistente en este contexto.

Para dar respuesta a esas preguntas, tomaremos como base el TFG de matemáticas titulado «Homología persistente como herramienta de análisis de redes neuronales» (véase [8]). En él expusimos los aspectos teóricos relacionados con el cálculo de la homología persistente en redes neuronales y tuvimos la ocasión de desarrollar una serie de algoritmos para su computación. En el presente trabajo trataremos de mejorar tales algoritmos y analizaremos su posible utilidad.

A continuación, a modo de resumen, describimos el cálculo de los complejos simpliciales a partir de una red neuronal.

1.1 El problema

Supondremos una red neuronal prealimentada, cuya estructura subyacente resulta en un grafo dirigido acíclico. Sobre esta red definimos la *importancia* entre neuronas directamente conectadas mediante la siguiente relación:

$$R_{ij} = \begin{cases} 1 & \text{si } i = j, \\ w_{ij}^{+} / \sum_{k,k\neq j} w_{kj}^{+} & \text{si } i \neq j, \end{cases} \tag{1.1.1}$$

donde $w_{ij}^{+} := \max\{w_{ij}, 0\}$ y w_{ij} es el peso entre la neurona i y j.

A partir de la anterior relación extendemos la definición de importancia entre cualquier par de neuronas multiplicando las importancias de las neuronas intermedias del camino entre la neurona de salida y la de llegada. Denotamos la noción de *importancia extendida* por $\overline{R_{ij}}$, y mediante ella podemos definir los p-símplices asociados a una red neuronal de la siguiente manera:

$$\nu_p^t = \begin{cases} \nu_0 & \text{si } p = 0, \\ \{\{u_{a_0}, \ldots, u_{a_p}\} \mid u_{a_i} \in \nu_0,\ \overline{R_{a_i a_j}} \geq t,\ \forall a_i > a_j\} & \text{si } p \geq 1, \end{cases}$$

donde cada u_{a_i} pertenece al nivel a_i del DAG subyacente a la red, $a_0 < \cdots < a_i < \cdots < a_p$, y $0 \leq t \leq 1$ es un parámetro real.

Ahora, considerando la unión de los p-símplices para un t fijo obtenemos un complejo simplicial abstracto, y tomando una sucesión monótona decreciente de *umbrales* t construimos un complejo simplicial filtrado. Para más detalles véase [8].

El software a desarrollar en el presente trabajo debe ser capaz de tomar una red neuronal prealimentada y calcular su complejo simplicial filtrado dada una colección de umbrales, tal y como acabamos de describir. A partir del complejo simplicial filtrado, delegará el cálculo de la homología persistente en una librería especializada y ofrecerá al usuario los diagramas que correspondan.

Capítulo 2
Planificación

A lo largo de este capítulo vamos a detallar los aspectos organizativos más relevantes del presente proyecto, entre los que se encuentran el alcance, la planificación temporal, los recursos y la metodología seguida.

2.1 Alcance

Dada la naturaleza del presente trabajo, la labor de clientes la ejercen los dos directores, que han delineado los requisitos y el alcance del proyecto. A continuación detallamos los requisitos del mismo.

2.1.1 Requisitos

El resultado del presente proyecto ha de ser un paquete de software bien documentado que permita realizar un análisis de redes neuronales mediante homología persistente. En la Tabla 2.1 podemos ver los requisitos fijados para el proyecto.

Requisitos del software	
Código	Requisito
COD-01	El software permitirá analizar redes neuronales siguiendo la interpretación expuesta en [12].
COD-02	El software tendrá un diseño modular que facilite su escalado y reutilización.
COD-03	El software tendrá un tiempo de ejecución aceptable.
COD-04	El software contará con su correspondiente documentación.
COD-05	El software vendrá acompañado con un dosier de experimentos que documente la aplicación de la homología persistente a redes neuronales.

Tabla 2.1: Requisitos del software.

2.1.2 Exclusiones

En la Tabla 2.2 podemos ver los requisitos que quedan excluidos del presente proyecto.

2.1.3 Entorno de distribución

Todo el software y los entregables asociados estarán disponibles de manera pública en el siguiente repositorio de *GitHub*: `https://github.com/joros244/TFGCompSci2023`.

Exclusiones	
Código	Requisito
RE-01	El software estará implementado en un lenguaje distinto de *Python*.
RE-02	El software se podrá aplicar sobre redes neuronales con alguna capa no completamente conectada.
RE-03	El software utilizará la librería *Ripser* para el cálculo de la homología persistente.

Tabla 2.2: Exclusiones.

2.1.4 Entregables

En la Tabla 2.3 y en la Tabla 2.4 recogemos los entregables que se contemplan para el proyecto. Para cada entregable incluimos su título, un código y una breve descripción.

Entregables relacionados con la gestión del proyecto		
Código	Título	Descripción
EG-01	Memoria	Es el presente documento. En él se recoge la planificación, el análisis, el diseño, la experimentación y el seguimiento y control del proyecto. También incluye los pasos seguidos durante el desarrollo.
EG-02	Presentación	Transparencias a usar el día de la defensa del trabajo.

Tabla 2.3: Entregables de gestión.

Entregables relacionados el producto		
Código	Título	Descripción
EP-01	Software	Código desarrollado para el proyecto.
EP-02	Documentación	Documentación asociada al software desarrollado.

Tabla 2.4: Entregables de producto.

2.1.5 EDT

La Figura 2.1 recoge la EDT del proyecto, y en la Tabla 2.5 podemos ver la asociación entre los paquetes de trabajo y los entregables.

2.2 Planificación temporal

En esta sección detallamos los aspectos de organización temporal del proyecto. Hacemos notar que el presente trabajo se ha realizado en el contexto de una beca de iniciación a la investigación, y que por tanto, el régimen de dedicación del mismo ha sido de aproximadamente tres horas diarias durante el período que comprende la beca: desde febrero hasta julio.

Figura 2.1: Estructura de descomposición del trabajo del proyecto.

Relación entre paquetes de trabajo y entregables	
Paquete	Entregable
1.1 Alcance 1.2 Diagrama Gantt 1.3 Calendario 2.1 Estudio 2.2 Diagrama de clases 4.1 Estudio 5. Memoria 6. Seguimiento y control 7. Reuniones	EG-01
3.1 Solución producida	EP-01, EP-02
8.1 Presentación	EG-02

Tabla 2.5: Asociación entre paquetes de trabajo y entregables.

2.2.1 Diagrama Gantt

En la Figura 2.2 y en la Figura 2.3 podemos ver el diagrama Gantt del proyecto. En él podemos ver los períodos de realización aproximados de los paquetes de trabajo.

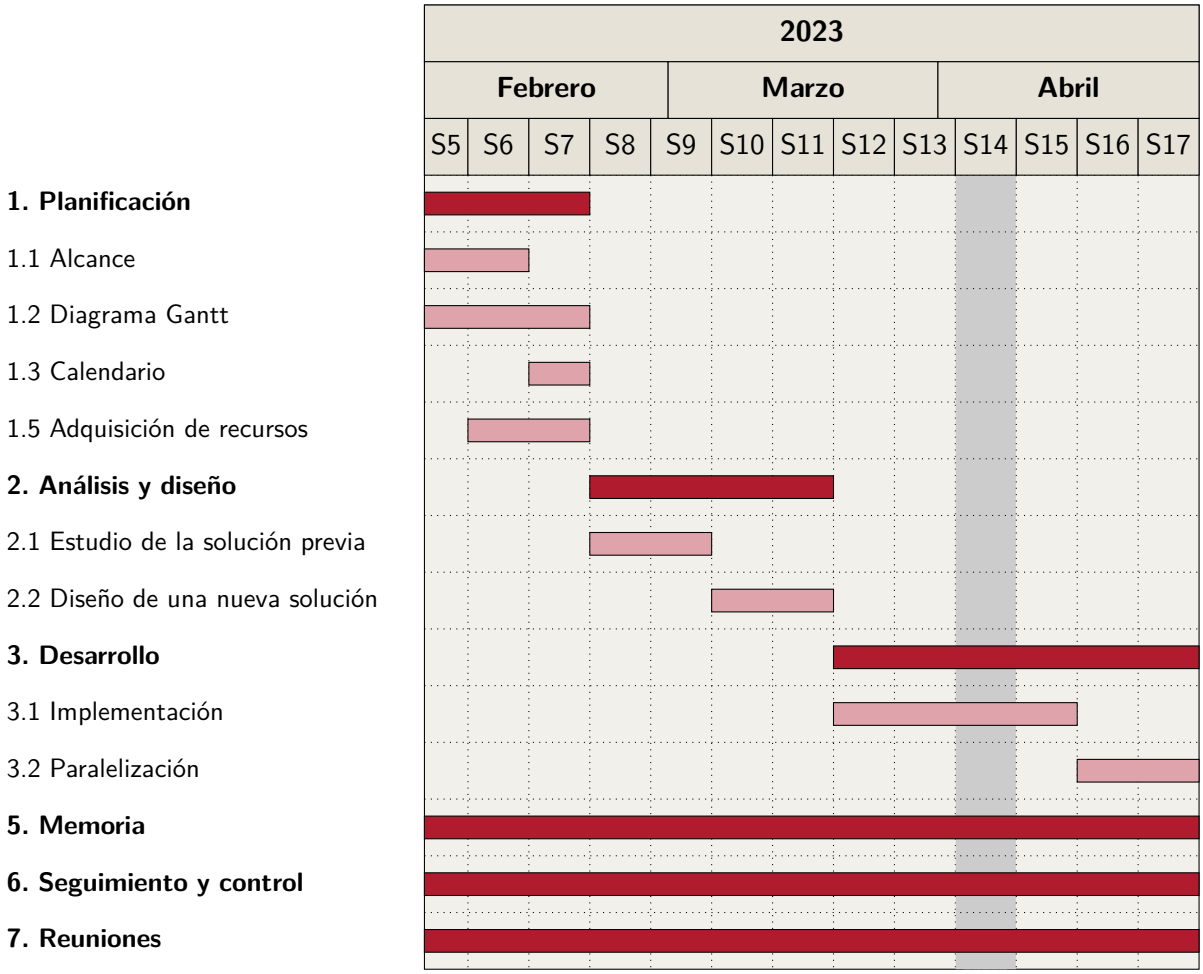

Figura 2.2: Diagrama Gantt parte 1.

2.2.2 Calendario e hitos

Para facilitar la visualización del ciclo de vida del proyecto, en la Figura 2.4 mostramos un calendario con los hitos más importantes del mismo. A excepción del principio y fin del proyecto, el resto de fechas son aproximaciones tomadas a partir del diagrama Gantt visto anteriormente. A continuación, detallamos las fechas más importantes que hemos destacado en el calendario:

- 1 de febrero: inicio del TFG.
- 17 de febrero: cierre de la planificación.
- 16 de marzo: cierre del análisis y diseño.
- 28 de abril: cierre del desarrollo.

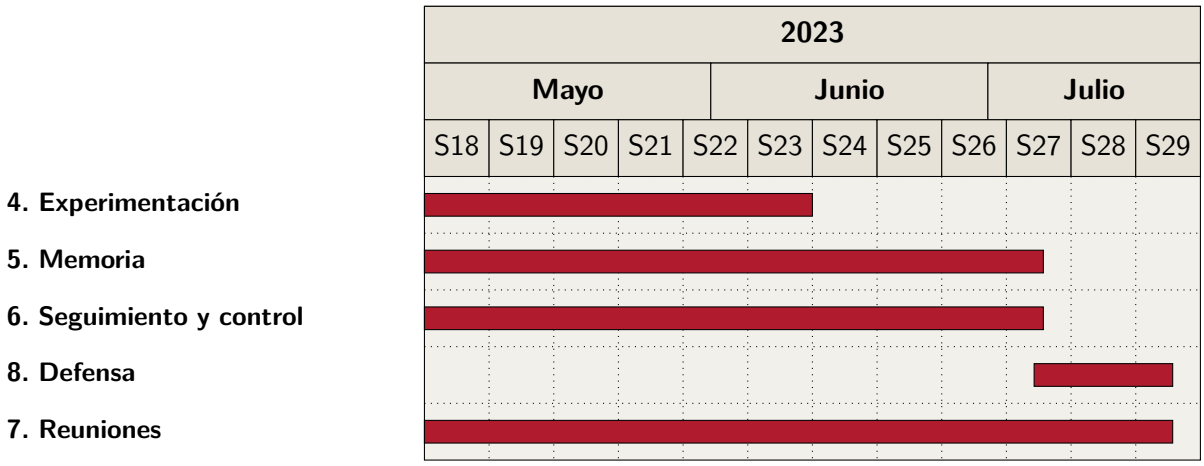

Figura 2.3: Diagrama Gantt parte 2.

- 8 de junio: cierre de la experimentación.
- 6 de julio: cierre de la memoria.
- 20 de julio: cierre del TFG.

2.3 Recursos computacionales

En la Tabla 2.6 podemos ver los recursos computacionales involucrados en el desarrollo del presente trabajo. Entre ellos destacamos el recurso RE-03, que ha resultado clave para las fases de desarrollo y experimentación.

Recursos computacionales del proyecto	
Código	Recurso
RE-01	Repositorio de *GitHub* de desarrollo.
RE-02	Repositorio de *GitHub* de producción.
RE-03	Acceso al servidor de cálculo *Simba* cedido por el grupo de investigación en Informática (*Psycotrip*).
RE-04	Entorno de *Python* con las librerías necesarias.

Tabla 2.6: Recursos computacionales.

2.4 Metodología

Para el desarrollo del presente trabajo plantemos el uso de una metodología *iterativa*. Esta metodología emerge de manera natural dadas las características del proyecto, y nos va a permitir mejorar el software con cada iteración hasta lograr el rendimiento buscado. La idea con cada iteración será

Figura 2.4: Calendario del proyecto con los hitos más importantes.

tratar de mejorar el software y evaluar su rendimiento con algún ejemplo. De esta manera podremos comparar el desempeño de las mejoras e ir seleccionando las mejores estrategias.

Capítulo 3
Análisis y diseño

En este capítulo vamos a analizar el código producido en [8], que nos servirá como base para el software que queremos desarrollar.

3.1 Análisis

Como ya hemos mencionado, vamos a trabajar sobre el proceso del cálculo del complejo simplicial abstracto asociado a una red neuronal. A continuación describimos formalmente el problema a resolver.

3.1.1 Descripción del problema

El software tomará como entrada una red neuronal cuya estructura subyacente será un grafo dirigido acíclico, y un umbral t. A partir de la red y sus pesos se realiza un preprocesado utilizando la definición de *importancia* (véase 1.1.1) y su extensión, obteniendo así una matriz cuadrada de tamaño n. Hacemos notar que este preprocesado no es significativo para el tiempo de ejecución del software, ya que en el peor caso conlleva un coste cuadrático. Así pues, centraremos nuestros esfuerzos en mejorar el proceso encargado del cálculo del complejo simplicial abstracto asociado a la red. La matriz obtenida del preprocesado, M, y el umbral t son las entradas de dicho proceso. A continuación, describimos más detalladamente tales entradas y la salida del proceso:

3.1.1.1 Entradas

- **Matriz M:** será una matriz triangular inferior de tamaño n, donde n será el número de neuronas de la red, y cuyas entradas de la diagonal son iguales a 1. Además, para todo a_{ij} de M se debe verificar que $0 \leq a_{ij} \leq 1$, y $\sum_{i=j+1}^{n-1} a_{ij} = 1$ para todo $j < n-1$. Finalmente, de M se debe poder deducir un grafo nivelado con saltos únicamente entre un nivel y el siguiente.

- **Umbral t:** será un parámetro real en el intervalo $[0, 1]$.

3.1.1.2 Salida

- **Complejo simplicial abstracto ν^t:** será una lista que verifique la definición de complejo simplicial abstracto junto con la restricción impuesta por el umbral t: la importancia entre el vértice «origen» y el vértice «destino» de cada elemento de la lista será mayor o igual que t.

Descritas las entradas y salida del proceso, pasamos a hacer una revisión de los algoritmos propuestos en [8].

3.1.2 Algoritmo propuesto por S. Watanabe y H. Yamana

En este apartado analizaremos el algoritmo original para el cálculo del complejo simplicial dada una red neuronal prealimentada (descrita por su correspondiente matriz M). Este algoritmo está propuesto en el artículo (véase [12]) que fundamenta el trabajo fin de grado de matemáticas, y puede verse en el Algoritmo 1.

Algoritmo 1: Algoritmo de S. Watanabe y H. Yamana.

```
Function GETSIMPLEX(matrix, currentPath, threshold):
    relevance ← 1.0, result ← ∅, origin ← currentPath[0];
    foreach destination in currentPath do
        relevance ← relevance × matrix[origin][destination];
        origin ← destination;
    end
    if relevance ≥ threshold then
        result.append(comb(currentPath));
        lastVertex ← currentPath[|currentPath| − 1];
        for i = 0 to n − 1 do
            if matrix[lastVertex][i] > 0 and i ≠ lastVertex then
                copyPath ← deepcopy(currentPath);
                copyPath.append(i);
                recursiveResult ← getSimplex(matrix, copyPath, threshold);
                foreach simplex in recursiveResult do
                    result.append(comb(simplex))
                end
            end
        end
    end
return unique(result)
```

A continuación, para facilitar su estudio, describimos de manera general los pasos seguidos en el Algoritmo 1:

1. A partir de la matriz M y una lista de vértices se calcula la importancia entre el primer y último vértice de la lista.

2. Si la importancia previamente calculada supera el umbral t, se añade al resultado el «conjunto potencia» del símplice constituido por la lista de vértices.

3. Ahora es necesario verificar si el símplice anteriormente añadido es maximal, en caso de serlo habremos terminado.

4. En caso de no serlo, se añade un nuevo vértice a la lista de vértices y se realiza una llamada recursiva. Tras la finalización de la llamada recursiva se toman los símplices devueltos y se añade su correspondiente «conjunto potencia» al resultado.

5. Finalmente se devuelve la lista resultado sin símplices repetidos. Esta lista contiene aquellos símplices que superan el umbral y cuyo origen es el primer vértice de la lista de vértices tomada en 1. Para obtener el complejo simplicial deseado será necesario iterar estos pasos por todos los vértices.

Observando el Algoritmo 1 podemos apreciar el uso de una función auxiliar llamada *comb()*. Esta función recibe una lista como entrada y devuelve su «conjunto potencia» (sin el vacío) en forma de lista. Hacemos notar que esta función representa uno de los cuellos de botella en el tiempo de ejecución del algoritmo, con un coste asintótico de $O(l \cdot 2^{l-1})$ donde l es el número de niveles de la red.

Finalmente, del estudio del Algoritmo 1 podemos deducir que el algoritmo realiza un «recorrido» en profundidad sobre el grafo dirigido acíclico (DAG por sus siglas en inglés). Este recorrido tiene como objetivo hallar la familia de caminos (símplices) maximales cuyo origen sea el primer vértice de la lista *puntos* y, hallados tales caminos, añadir los correspondientes subsímplices (calculados con *comb()*) al resultado. Nótese que el recorrido en profundidad representa otro de los cuellos de botella en el tiempo de ejecución del algoritmo que, además, debemos ejecutar sobre cada vértice del DAG.

3.1.3 Algoritmo propio desarrollado en el TFG de matemáticas

A continuación vamos a analizar el algoritmo propio que se propuso durante el desarrollo del TFG de matemáticas (véase [8]). Este algoritmo se desarrolló como consecuencia de varias dificultades durante el estudio de [12] y el uso del Algoritmo 1. En primer lugar, mostramos su descripción que puede verse en el Algoritmo 2.

Algoritmo 2: Algoritmo TFGM

1. Cálculo de los símplices maximales:
   (a) Construcción del DAG asociado a la matriz M filtrada eliminando las entradas que no superan t.
   (b) Construcción del DAG clausura transitiva del DAG previamente calculado:
      i. Construcción del grafo clausura transitiva mediante el uso de los algoritmos que pueden encontrarse en [9].
      ii. Construcción de la matriz asociada al DAG clausura transitiva completando adecuadamente M filtrada.
   (c) A partir del grafo clausura transitiva y su matriz asociada se recorre M para obtener los complejos simpliciales maximales comprobando que superan el umbral t.
   (d) Devolver la lista de símplices maximales resultante.
2. Añadir el «conjunto potencia» de cada símplice maximal a la lista resultado.

Observando el Algoritmo 2 podemos identificar algunas diferencias y semejanzas con respecto al Algoritmo 1. Las diferencias entre ambos se derivan del uso del DAG en el segundo algoritmo, y en lo referente a sus semejanzas, observamos que ambos presentan los mismos procesos críticos: el «recorrido» en profundidad y el cálculo del conjunto potencia (paso 2). Así pues, centraremos nuestra atención en mejorar la eficiencia de estos pasos, además de eliminar una serie de redundancias presentes en ambos algoritmos.

3.1.4 Análisis de la complejidad computacional

A fin de completar el análisis de ambos algoritmos vamos a realizar un estudio de la complejidad computacional asintótica de los mismos. En concreto, estudiaremos el coste del «recorrido en profundidad» presente en ambos algoritmos, ya que es el proceso crítico con mayor potencial de mejora.

Para estudiar la complejidad supondremos el peor escenario posible: una red completamente conectada de n neuronas distribuidas uniformemente a lo largo de l capas, y un umbral $t = 0$. Nótese que este escenario es altamente improbable en la realidad, pues el umbral t supone un colapso total de las clases de homología subyacentes a la red, lo que resulta en unos diagramas de persistencia poco informativos. No obstante, este escenario nos ofrece una cota del rendimiento de los algoritmos que nos servirá como referencia de cara a la fase de desarrollo.

Complejidad del Algoritmo 1. Tenemos que cada vértice, excluyendo la primera capa, se recorre implícitamente $(\frac{n}{l})^{l_i}$ veces, donde l_i es el nº de capa del vértice. También tenemos que en cada capa se recorren explícitamente $(n + l_i + 1)\frac{n}{l}$ vértices. Así pues, haciendo una traza del recorrido sobre todos los vértices, tenemos que se recorren un total de:

$$\sum_{j=0}^{l-1}\sum_{k=1}^{l-j}(n+k)(\frac{n}{l})^k \text{ vértices.}$$

Es decir, tenemos un coste asintótico (incluyendo la función «conjunto potencia») total del orden de:

$$O(n \cdot (\frac{n}{l})^l(1 + l \cdot 2^{l-1})).$$

Complejidad del Algoritmo 2. Por una parte, tenemos que los costes relacionados con el cálculo del DAG y su clausura transitiva son del orden de $O(n^2)$. Por otra parte, tenemos que los costes asociados a completar la matriz y obtener los símplices maximales son del orden de $O(n(\frac{n}{l})^{l+1})$, lo que nos deja un coste (incluyendo la función «conjunto potencia») total aproximado del orden de:

$$O(n \cdot (\frac{n}{l})^{l+1}(1 + l \cdot 2^{l-1})).$$

A este coste asintótico mayor debemos añadirle el efecto de una serie de redundancias que hacen que el Algoritmo 2 sea significativamente más ineficiente que el Algoritmo 1.

Nota. Destacamos que, aunque los costes computacionales son muy elevados, el coste del «recorrido en profundidad» es un coste polinómico en términos del número de neuronas de la red (n). Sin embargo, notemos que el coste de la función «conjunto potencia» es del orden de:

$$O(l \cdot 2^{l-1}),$$

que es el factor responsable de disparar los costes a medida que se añaden capas a la red.

3.1.5 Comparativa experimental de ambos algoritmos

Para finalizar el análisis, vamos a comparar el tiempo de ejecución de ambos algoritmos. Para ello vamos a estudiar su desempeño sobre una red neuronal de tres capas completamente conectadas (FCN por sus siglas en inglés) con una disposición de $(300, 100, 10)$ neuronas, entrenada sobre el «dataset» MNIST. En cuanto la colección de umbrales, consideraremos el siguiente conjunto:

$r = \{1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 1.0 \times 10^{-1}, 0.9 \times 10^{-1}, 0.8 \times 10^{-1}, 0.7 \times 10^{-1}, 0.6 \times 10^{-1}, 0.5 \times 10^{-1}, 0.4 \times 10^{-1}, 0.3 \times 10^{-1}, 0.2 \times 10^{-1}, 1.0 \times 10^{-2}, 0.9 \times 10^{-2}, 0.8 \times 10^{-2}, 0.7 \times 10^{-2}, 0.6 \times 10^{-2}, 0.5 \times 10^{-2}, 0.4 \times 10^{-2}, 0.3 \times 10^{-2}, 0.2 \times 10^{-2}, 1.0 \times 10^{-3}, 0.9 \times 10^{-3}, 0.8 \times 10^{-3}, 0.7 \times 10^{-3}, 0.6 \times 10^{-3}, 0.5 \times 10^{-3}, 0.4 \times 10^{-3}, 0.3 \times 10^{-3}, 0.2 \times 10^{-3}, 1.0 \times 10^{-4}, 0.9 \times 10^{-4}, 0.8 \times 10^{-4}, 0.7 \times 10^{-4}, 0.6 \times 10^{-4}, 0.5 \times 10^{-4}, 0.4 \times 10^{-4}, 0.3 \times 10^{-4}, 0.2 \times 10^{-4}, 1.0 \times 10^{-5}, 0.9 \times 10^{-5}, 0.8 \times 10^{-5}, 0.7 \times 10^{-5}, 0.6 \times 10^{-5}, 0.5 \times 10^{-5}, 0.4 \times 10^{-5}, 0.3 \times 10^{-5}, 0.2 \times 10^{-5}, 1.0 \times 10^{-6}, 0.9 \times 10^{-6}, 0.8 \times 10^{-6}, 0.7 \times 10^{-6}, 0.6 \times 10^{-6}, 0.5 \times 10^{-6}, 0.4 \times 10^{-6}, 0.3 \times 10^{-6}, 0.2 \times 10^{-6}, 1.0 \times 10^{-7}\}$

Dada que esta configuración fue la escogida por los autores en [12], la mantendremos para futuras comparaciones. A continuación, veamos una comparativa del desempeño de los algoritmos anteriormente descritos, donde el algoritmo de los autores (Algoritmo 1) se corresponde con WY Mk.I y el algoritmo TFGM (Algoritmo 2) con TFG Mk.I. Como hemos comentado previamente, el algoritmo TFG Mk.I posee varias redundancias que lo harán más lento.

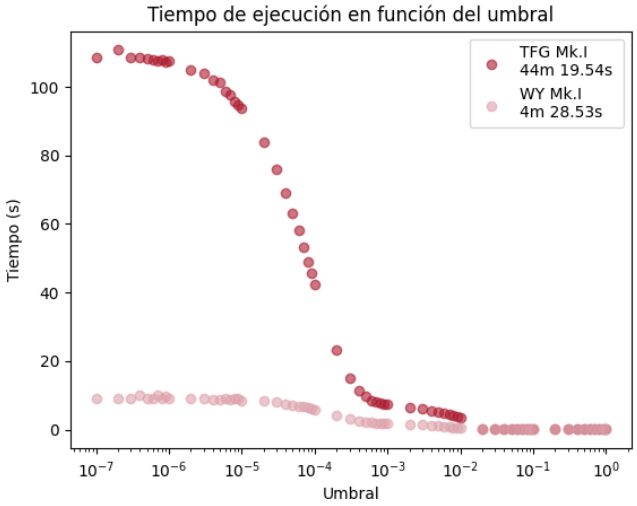

Figura 3.1: Comparativa de los algoritmos iniciales WY Mk.I y TFG Mk.I.

De la Figura 3.1 podemos observar que:

- A medida que aumenta el umbral, disminuye el tiempo de ejecución. Esto es debido a que un menor umbral permite la aparición de más símplices, lo que afecta significativamente al tiempo de ejecución.
- TFG Mk.I resulta mucho más ineficiente que WY Mk.I, habiendo una gran diferencia de tiempo entre ambos.

Con esta comparativa, que nos ofrece una visión más clara de los algoritmos a mejorar, finalizamos esta sección de análisis. Nótese que en este análisis hemos omitido el estudio de la corrección de estos algoritmos, pues quedó suficientemente revisada en [8]. A continuación, abordamos el diseño del software, centrado en facilitar la mejora de los algoritmos y la extensibilidad del software.

3.2 Diseño

Comenzamos el diseño del software con el diagrama de flujo que podemos ver en la Figura 3.2. En este diagrama se recoge una idea general del software a construir, así como los principales subprocesos del mismo.

Durante la fase de implementación centraremos nuestros esfuerzos en los dos primeros bloques, delegando el tercero en librerías externas. En concreto, las librerías escogidas para el cálculo de los diagramas de persistencia son *Dionysus* (véase [7]) y *GUDHI* (véase [10]). Estas fueron las librerías que se emplearon en el trabajo de matemáticas y vamos a mantener su uso en el presente trabajo.

Figura 3.2: Diagrama de flujo del software.

Hacemos notar que aunque el diseño se ha realizará asumiendo el uso de estas librerías, trataremos de desacoplar su funcionalidad para poder adaptar fácilmente al uso de otras librerías si en el futuro fuese necesario.

Dada la naturaleza del software que se pretende construir, proponemos el diseño en clases que puede verse en la Figura 3.3. Este diseño está centrado en la extensibilidad del software, de manera que podamos añadir más funcionalidad y ajustarla a nuestras necesidades a medida que tratamos de mejorar los algoritmos. Para la elaboración del diagrama hemos empleado la herramienta *Modelio* (véase [1]).

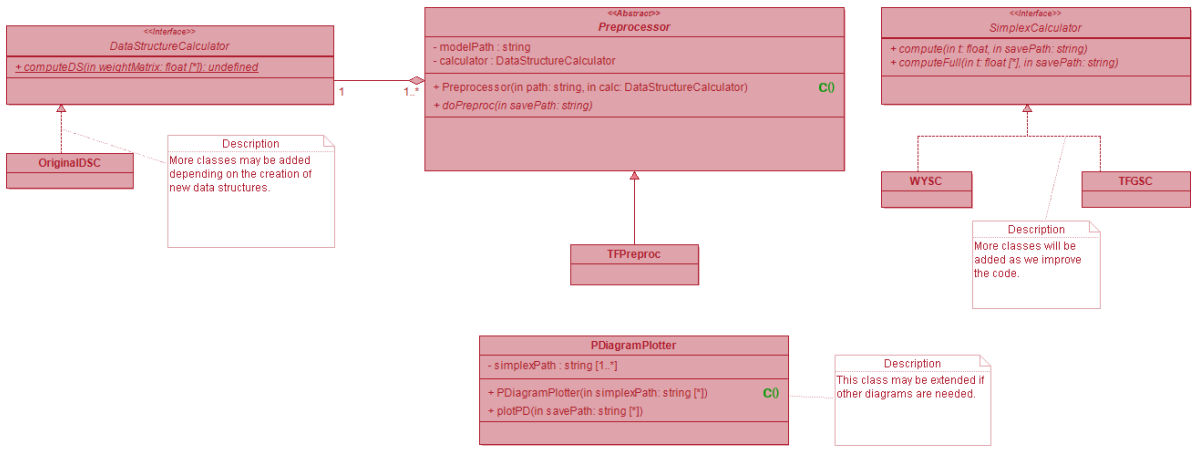

Figura 3.3: Diagrama de clases.

De la observación del anterior diagrama se pueden deducir algunas decisiones de diseño que destacamos a continuación:

- El resultado del preprocesado de la red (estructura de datos) se almacenará como un fichero en el disco. Esta decisión está motivada por la reutilización de la estructura de datos, ya que una vez calculada podremos utilizarla tantas veces como deseemos en el cálculo de los símplices.
- El tipo de la estructura de datos es indefinido, es decir, lo consideramos un tipo abstracto de datos. Esta decisión se deriva de la ineficiencia del uso de matrices para nuestro problema, de modo que dejamos abierta la posibilidad de cambiar la estructura de datos.
- El resultado del procesado de la estructura de datos (complejo simplicial) se almacenará como un fichero en el disco. Esto se debe a que el complejo simplicial podría llegar a ser muy pesado en memoria, además de conllevar un gran tiempo de cálculo. Por lo tanto, tener el complejo almacenado en disco facilitará su reutilización y la conservación de evidencias. Hacemos notar que en los ejemplos estudiados, el tamaño del complejo simplicial «serializado» y almacenado en disco varía entre 20MB y 500MB, dependiendo de la red estudiada.

- El objeto encargado de calcular los diagramas de persistencia («*PDiagramPlotter*») podrá recibir varios complejos simpliciales y podrá producir (guardar en disco) varios diagramas de persistencia. Esta decisión está motivada por un escenario en el que el usuario haya procesado varias redes y quiera calcular los diagramas de persistencia sin la necesidad de crear varios objetos.
- Las implementaciones de «*Simplex Calculator*» dependerán de la implementación de la correspondiente estructura de datos. Esta decisión está motivada por cuestiones de eficiencia, aunque hacemos notar que el presente diseño permite añadir fácilmente una capa de abstracción sobre la estructura de datos.

Capítulo 4
Desarrollo

En este capítulo detallaremos la fase de desarrollo del proyecto. En concreto, tomaremos los algoritmos vistos en la fase de análisis y trataremos de mejorarlos para hacer que sean, al menos, utilizables para calcular el complejo simplicial asociado a una red neuronal. Una vez implementadas las mejoras, evaluaremos los nuevos algoritmos siguiendo el ejemplo visto en la Subsección 3.1.5, de manera que podamos observar con mayor claridad la escala de las mejoras.

4.1 Implementación de nuevas soluciones

A lo largo de esta sección vamos a proceder con la mejora iterativa de los algoritmos, realizando tantos refinamientos de los mismos como sean necesarios para reducir al mínimo posible su tiempo de ejecución. Como ya identificamos durante la fase de análisis, los principales cuellos de botella de los algoritmos son el «*recorrido en profundidad*» y la función «*conjunto potencia*», por lo que serán el foco de nuestros esfuerzos durante los intentos de mejora.

Por otra parte, obviaremos cualquier mejora realizable en el preprocesado de la red, si bien, como ya mencionamos durante la fase de diseño, dejamos abierta la posibilidad de cambiar la estructura de datos si fuese necesario. Finalmente, hacemos notar que cada algoritmo desarrollado estará encapsulado en su correspondiente clase que implementará la interfaz «*SimplexCalculator*», tal y como estipula el diagrama visto en la Figura 3.3.

4.1.1 Primer intento de mejora: algoritmos WY Mk.II y TFG Mk.II

4.1.1.1 Algoritmo WY Mk.II

Tras el estudio del Algoritmo 1, podemos identificar los siguientes cambios que mejorarán el rendimiento del mismo:

1. Modificación de la condición de permanencia de un bucle «for» para recorrer únicamente las entradas de la matriz M (triangular inferior) que contienen la información relevante. Véase «Cambio 1» en el Algoritmo 3.

2. Eliminación de una de las condiciones de la sentencia «if», por ser redundante como consecuencia de «Cambio 1». Véase «Cambio 2» en el Algoritmo 3.

3. Eliminación de la función «comb», por ser redundante, al añadir los símplices calculados por la recursividad. Véase «Cambio 3» en el Algoritmo 3.

Estas modificaciones, como veremos más adelante, logran reducir el tiempo de ejecución pasando de unos pocos minutos a segundos. En la Tabla 4.1 mostramos el coste computacional del algoritmo así como el desempeño de su implementación.

Algoritmo 3: Algoritmo de los autores modificado: WY Mk.II.

```
Function GETSIMPLEX(matrix, currentPath, threshold):
    relevance ← 1.0, result ← ∅, origin ← currentPath[0];
    foreach destination in currentPath do
        relevance ← relevance × matrix[origin][destination];
        origin ← destination;
    end
    if relevance ≥ threshold then
        result.append(comb(currentPath));
        lastVertex ← currentPath[|currentPath| − 1];
        for i = 0 to lastVertex do                                  /* Cambio 1 */
            if matrix[lastVertex][i] > 0 then                       /* Cambio 2 */
                copyPath ← deepcopy(currentPath);
                copyPath.append(i);
                recursiveResult ← getSimplex(matrix, copyPath, threshold);
                foreach simplex in recursiveResult do
                    result.append(simplex)                          /* Cambio 3 */
                end
            end
        end
    end
return unique(result)
```

4.1.1.2 Algoritmo TFG Mk.II

Como hemos observado anteriormente, el algoritmo propuesto en el Algoritmo 2 tiene pasos redundantes. Así pues, como primer intento de mejora vamos a eliminar el paso b) ii), con lo que obtenemos un nuevo algoritmo que puede verse en el Algoritmo 4.

Algoritmo 4: Algoritmo propio modificado: TFG Mk.II

1. Cálculo de los símplices maximales:
   (a) Construcción del DAG asociado a la matriz M filtrada eliminando las entradas que no superan t.
   (b) Construcción del DAG clausura transitiva del DAG previamente calculado:
      i. Construcción del grafo clausura transitiva mediante el uso de los algoritmos que pueden encontrarse en [9].
   (c) A partir del grafo clausura transitiva y su matriz asociada se recorre M para obtener los complejos simpliciales maximales comprobando que superan el umbral t.
   (d) Devolver la lista de símplices maximales resultante.
2. Añadir el «conjunto potencia» de cada símplice maximal a la lista resultado.

Con esta modificación, hemos eliminado una de las redundancias, ganando así algo de tiempo. No obstante, como puede verse en la Tabla 4.1, esta mejora no es muy significativa.

4.1.2 Segundo intento de mejora: algoritmo TFG Mk.III

A pesar de haber eliminado una redundancia, el algoritmo TFG Mk.II sigue siendo demasiado lento frente a WY Mk.II. Por ello, en esta sección vamos a eliminar todas las redundancias posibles del algoritmo presentado en el Algoritmo 2, además de afinar la implementación del cálculo del «conjunto

potencia» (función *comb*) siguiendo para ello el ejemplo proporcionado por la implementación de los autores de esta función.

4.1.2.1 Algoritmo TFG Mk.III

En el Algoritmo 5 podemos ver la propuesta del nuevo algoritmo.

Algoritmo 5: Algoritmo TFG Mk.III

1. Cálculo del DAG asociado a M.
2. Recorrer el DAG consultando las entradas de M para cada par de vértices relacionados del DAG.
3. Si $a_{ij} \geq t$ con $a_{ij} \in M$, entonces ejecutar un recorrido en profundidad, con origen la arista $\{i, j\}$, para hallar cada símplice maximal, s, del que $\{i, j\}$ es cara.
4. Hallados los símplices maximales, añadir el «conjunto potencia» de cada uno de ellos al resultado.

Hacemos notar que el Algoritmo 5 resulta mucho más eficiente para el cálculo del complejo simplicial buscado. Esta ganancia radica principalmente en:

- **Uso del DAG:** aunque sea necesario realizar el cálculo previo del DAG con coste cuadrático, para el ejemplo estudiado este cálculo se realiza en un tiempo insignificante, y reporta una ganancia importante: evitamos recorrer entradas nulas de M. Esta optimización, como veremos, hace que TFG Mk.III resulte más rápido que WY Mk.II.
- **Optimización de la función «conjunto potencia»:** tras una serie de comprobaciones, observamos que la función «conjunto potencia» implementada en TFG Mk.I y TFG Mk.II representaba un cuello de botella importante en toda la ejecución. Haciendo uso de sentencias presentes en la implementación de los autores se ha conseguido reducir aún más el tiempo de ejecución de TFG Mk.III.

Observamos que, aprovechando la lista de adyacencia, el coste asintótico se reduce dos órdenes de magnitud, haciendo que el Algoritmo 5 sea más eficiente que el Algoritmo 3. No obstante, hacemos notar que en el Algoritmo 5 siempre se calculará el DAG asociado con su correspondiente coste cuadrático. Para más detalles sobre el coste computacional de este algoritmo, y de su desempeño en el ejemplo estudiado, véase la Tabla 4.1.

4.1.3 Nueva estructura de datos y algoritmo TFG Mk.IV

A la vista de la mejora obtenida con TFG Mk.III, vamos a trasladar el foco de mejora del algoritmo a la estructura de datos. Así pues, planteamos una nueva estructura de datos que nos permitirá reducir aún más el tiempo de ejecución.

4.1.3.1 Nueva estructura de datos

Planteamos una estructura de datos muy similar a la lista de listas que implementaba M: consideramos un vector de tamaño n, donde la componente i-ésima es un vector *ordenado* de 2-tuplas de tamaño k_i (el número de vértices adyacentes a i). La primera componente de la 2-tupla será el número del vértice adyacente, y la segunda componente será el umbral t para viajar a ese vértice.

Es decir, tendremos un vector de n componentes donde cada componente es un diccionario ordenado cuyas claves son los vértices adyacentes y los valores son los umbrales t. El orden del vector de 2-tuplas (diccionario), de mayor a menor, viene dado por la segunda componente de la 2-tupla (umbral t).

Esta estructura de datos presenta el inconveniente de tener que ordenar n vectores (diccionarios) de tamaño k_i. Aunque para el ejemplo estudiado el tiempo de ejecución de este cálculo resulta insignificante, podría no serlo en ejemplos de mayor tamaño.

4.1.3.2 Algoritmo TFG Mk.IV

El algoritmo TFG Mk.IV queda descrito en el Algoritmo 6.

Algoritmo 6: Algoritmo TFG Mk.IV

1. Para cada vértice ejecutar un recorrido en profundidad aprovechando el orden del vector de adyacencia.
2. Cada vez que el recorrido llega a un vértice final (se ha encontrado un símplice maximal) registrar el símplice en el resultado.
3. Hallados los símplices maximales, añadir el «conjunto potencia» de cada uno de ellos al resultado.

Hacemos notar que en esta versión no se construye el DAG, por lo que nos ahorramos su coste computacional. Además, como el algoritmo aprovecha el orden de la nueva estructura de datos, el tiempo de ejecución disminuye aún más (véase la Tabla 4.1).

4.1.4 Usando el conjunto de umbrales: algoritmo TFG Mk.V

Hasta ahora hemos considerado como entrada en todas las funciones la matriz M (o la nueva estructura de datos) y un umbral arbitrario t. No obstante, si reflexionamos sobre el conjunto de umbrales r, observamos que existe un orden en el conjunto del que nos podemos beneficiar: calculando el complejo simplicial abstracto filtrado con el menor t podemos evitar la búsqueda en profundidad del resto de filtraciones. Esta idea es la que motiva el algoritmo TFG Mk.V.

4.1.4.1 Algoritmo TFG Mk.V

A partir de la idea anterior, el algoritmo recibirá como entradas el conjunto de umbrales r y la nueva estructura de datos d. A partir de estas entradas será capaz de, dado un t mayor que el menor umbral de r, producir el complejo simplicial filtrado correspondiente. En el Algoritmo 7 mostramos el nuevo algoritmo que implementa esta propuesta de mejora.

Observamos que, al ejecutar un único recorrido en profundidad, el tiempo de ejecución se reduce ligeramente (véase la Tabla 4.1). No obstante, hacemos notar que este algoritmo es muy intensivo en memoria, ya que supone almacenar la lista ordenada de símplices hasta la solicitud de la última filtración.

4.2 Algoritmos secuenciales: rendimiento y coste

En esta sección vamos a comparar el tiempo de ejecución de las implementaciones de los algoritmos secuenciales desarrollados. Para ello evaluaremos su desempeño de la misma manera que en la

Algoritmo 7: Algoritmo TFG Mk.V

1. A partir de d y del último umbral de r, ejecutar un recorrido en profundidad desde cada vértice. Este recorrido en profundidad será idéntico al ejecutado en TFG Mk.IV, con la salvedad de que debemos registrar en el resultado cada símplice por el que pasemos, así como su importancia asociada.
2. Ordenar el resultado obtenido del paso anterior en función de la importancia de cada símplice de mayor a menor.
3. Para cada t, menor o igual al último umbral de r, solicitado recorrer el resultado ordenado del paso anterior calculando el «conjunto potencia» de cada símplice.
4. Eliminar los repetidos originados en el paso anterior y devolver el resultado.

Comparativa de coste y desempeño		
Versión	Coste computacional	Tiempo de ejecución
WY Mk.I	$O(n \cdot (\frac{n}{l})^l(1 + l \cdot 2^{l-1}))$	4m 28.53s
TFG Mk.I	$O(n \cdot (\frac{n}{l})^{l+1}(1 + l \cdot 2^{l-1}))$	44m 19.54s
WY Mk.II	$O(\frac{n}{2} \cdot (\frac{n}{l})^l(1 + l \cdot 2^{l-1}))$	36.91s
TFG Mk.II	$O(n \cdot (\frac{n}{l})^{l+1}(1 + l \cdot 2^{l-1}))$	41m 31.87s
TFG Mk.III	$O((\frac{n}{l})^l(1 + l \cdot 2^{l-1}))$	6.97s
TFG Mk.IV	$O((\frac{n}{l})^l(1 + l \cdot 2^{l-1}))$	5.74s
TFG Mk.V	$O((\frac{n}{l})^l(1 + l \cdot 2^{l-1}))$	5.32s

Tabla 4.1: Comparativa de los algoritmos secuenciales.

Subsección 3.1.5. Completaremos dicha comparativa incluyendo el coste computacional de cada versión. En la Tabla 4.1 mostramos dicha información, incluyendo el coste y desempeño de la primera versión de los algoritmos.

Para analizar el coste computacional de los algoritmos hemos supuesto el peor escenario posible, y hemos procedido tal y como hemos visto en la fase de análisis. Finalmente, de manera gráfica, podemos ver una comparativa de los tiempos de ejecución en la Figura 4.1.

Finalmente, hacemos notar que, aunque es posible «linealizar» el recorrido en profundidad, hemos decidido omitirlo, pues según nuestras estimaciones el nuevo recorrido hubiera sido muy intensivo en operaciones de lectura y escritura en disco, lo que a largo plazo hubiera ralentizado la ejecución de todo el programa. Adicionalmente, destacamos que, aún haciendo lineal el recorrido en profundidad, el programa se vería igualmente afectado por la función «conjunto potencia».

4.3 Paralelización de las nuevas soluciones

Habiendo desarrollado un algoritmo secuencial lo suficientemente eficiente, en esta sección vamos a tratar de desarrollar un algoritmo paralelo para intentar reducir aún más el tiempo de ejecución y facilitar su escalado. Para paralelizar los algoritmos seguiremos el camino marcado por la paralelización de los datos.

Hacemos notar que la evaluación de los algoritmos paralelos se ha realizado con una ejecución de los cálculos en local, pues los tiempos obtenidos con la máquina *Simba* (cedida por el grupo

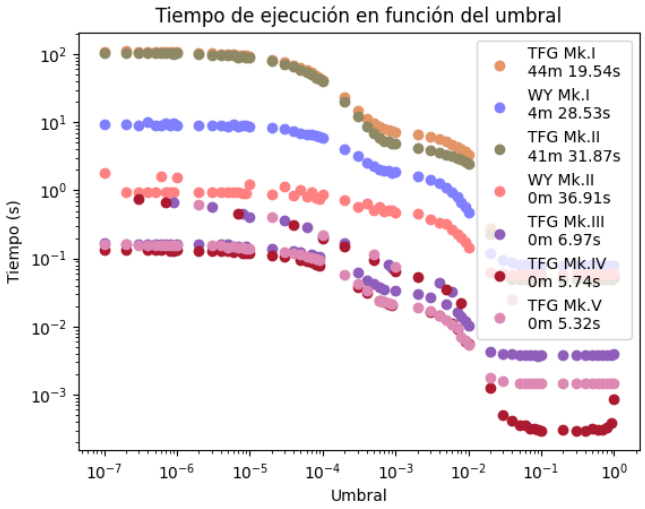

Figura 4.1: Comparativa de los tiempos de ejecución de los algoritmos (escala logarítmica).

Psycotrip) eran peores. De las comprobaciones realizadas dedujimos que este comportamiento estaba relacionado con las versiones de *Python* instaladas en cada ordenador. En concreto, en local contamos con *Python* 3.10, mientras que en *Simba* tenemos *Python* 3.9. La dificultad de una actualización de *Simba* nos condujo irremediablemente a evaluar los algoritmos en local, por lo que los valores obtenidos deben tomarse como una mera referencia de las posibilidades de ganancia de la ejecución paralela de los algoritmos.

4.3.1 Paralelización usando el conjunto de umbrales

La primera estrategia de paralelización de los datos consiste en utilizar una «*ThreadPool*» para ejecutar TFG Mk.V y WY Mk.II. Concretamente, emplearemos un hilo para cada umbral t de r. En la Figura 4.2 podemos ver una comparativa del desempeño de ambos algoritmos (P1) en su versión paralela.

Como podemos observar en la Figura 4.2, la versión paralela del algoritmo de los autores reporta poco o ningún beneficio frente a su versión secuencial. Sin embargo, la versión paralela de TFG Mk.V consigue disminuir ligeramente el tiempo de ejecución.

4.3.2 Paralelización usando los vértices

En esta estrategia, al igual que en la anterior, usaremos una «*ThreadPool*» para ejecutar los algoritmos TFG Mk.V y WY Mk.II. Ahora bien, en esta estrategia lanzaremos un hilo sobre cada vértice para hacer el correspondiente recorrido en profundidad.

Al igual que antes, en la Figura 4.2 podemos observar como la versión paralela del algoritmo de los autores no mejora el tiempo de ejecución de su versión secuencial. Por otra parte, destacamos el extraordinario desempeño de TFG Mk.V en su versión paralela, ostentando el menor tiempo de ejecución hasta ahora.

4.4 Algoritmos paralelos: eficiencia y rendimiento

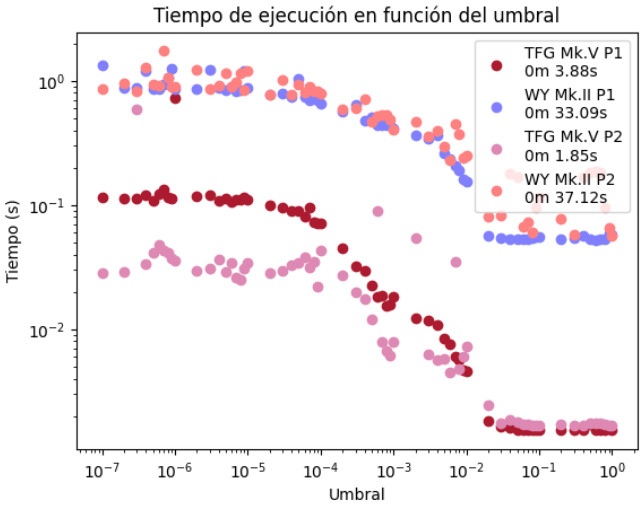

Figura 4.2: Comparativa de los tiempos de ejecución de los algoritmos paralelos (escala logarítmica).

Hacemos notar que la complejidad asintótica de los algoritmos paralelos permanece inalterada respecto de la versión WY Mk.II y TFG Mk.V respectivamente, pues las mejoras consisten en calcular las filtraciones de manera paralela y en calcular los símplices maximales asociados a cada vértice de manera paralela . No obstante, cabe destacar que la paralelización nos brinda un límite teórico de mejora de $\frac{t_s(n)}{p}$, donde p es el número de trabajadores (hilos en nuestro caso, véase [2, 4]) empleados, y $t_s(n)$ es el tiempo empleado por el mejor algoritmo secuencial conocido en la resolución del problema.

Tomando como referencia los experimentos anteriores y fijando $p = 12$, obtenemos un tiempo límite de $0.44s$ y $2.92s$ para los algoritmos TFG y WY respectivamente. Como podemos observar en la Figura 4.2 los tiempos reales se alejan bastante de los límites teóricos, si bien consiguen alguna mejora respecto de los algoritmos secuenciales.

A fin de analizar el valor de una ejecución paralela, vamos a estudiar una serie de métricas que reflejen la ganancia lograda.

La primera métrica a estudiar es el «*speed-up*», que mide la aceleración obtenida con una ejecución paralela. Esta métrica se define como el cociente $\frac{t_s(n)}{t_p(n)}$, donde $t_p(n)$ es el tiempo empleado por el mejor algoritmo paralelo conocido en la resolución del problema. Hacemos notar que esta métrica proporciona valores reales positivos no acotados, de manera que un valor muy elevado reflejaría una gran mejora con la ejecución paralela. Obsérvese que un valor de «speed-up» por debajo de 1 refleja una pérdida de tiempo con la ejecución paralela, y un valor superior a 1 refleja una ganancia de tiempo con la ejecución paralela.

Otra de las métricas a estudiar es la *eficiencia*, definida como el cociente entre el «speed-up» y el número de trabajadores. Nótese que la eficiencia es un valor real entre 0 y 1, donde una eficiencia de 1 reflejaría que los trabajadores realizan un trabajo útil durante todo el tiempo.

En la Tabla 4.2 podemos ver el valor de estas métricas para el caso que nos ocupa. Como podemos observar, en líneas generales la mejora lograda por la ejecución paralela es bastante pobre y, acudiendo al valor de la eficiencia, podemos ver que los trabajadores no están realizando un

trabajo útil durante la mayoría del tiempo. Este hecho está relacionado con la dificultad de producir código paralelo en *Python*, cuyos mecanismos de paralelización obligan a trabajar sobre memoria completamente separada (lo que añade tiempo adicional de inicialización), o a turnarse para acceder a la memoria («*Global Interpreter Lock*»).

Comparativa de desempeño y eficiencia				
Versión	Límite teórico	Ejecución real	Speed-up	Eficiencia
WY Mk.II P1	2.92s	33.09s	1.06	0.088
TFG Mk.V P1	0.44s	3.88s	1.37	0.114
WY Mk.II P2	2.92s	37.12s	0.95	0.079
TFG Mk.V P2	0.44s	1.85s	2.87	0.239

Tabla 4.2: Comparativa de los algoritmos paralelos.

Capítulo 5
Experimentación

Finalizado el desarrollo, en el presente capítulo vamos a estudiar el uso y significado de la homología persistente aplicada a redes neuronales. Para ello, tomaremos el software producido, lo ejecutaremos sobre una serie de ejemplos y analizaremos los resultados.

5.1 Experimentos

En la presente sección desarrollamos los experimentos realizados con el software producido. En concreto, nos proponemos analizar los diagramas de persistencia de dos familias de modelos: *CIFAR* y *MNIST*.

5.1.1 Modelos CIFAR

Se han entrenado tres modelos de visión por computador sobre el dataset CIFAR10 (6000 imágenes de 32x32x3 píxeles). La tarea a realizar por los modelos consiste en clasificar un conjunto de imágenes a color en 10 categorías distintas. Los tres modelos se dividen a su vez en una parte convolucional y otra completamente conectada. Para los propósitos del estudio hemos considerado la parte convolucional fija, y hemos cambiado la parte completamente conectada, pues actualmente el algoritmo solo es aplicable a redes con capas completamente conectadas (FCN por sus siglas en inglés). A continuación detallamos la arquitectura de la parte completamente conectada de los tres modelos:

- **Modelo CIFAR-256**: el modelo está constituido por tres capas completamente conectadas (FC por sus siglas en inglés) con una disposición de $(256, 256, 10)$ neuronas.
- **Modelo CIFAR-512**: el modelo está constituido por tres capas «FC» con una disposición de $(512, 512, 10)$ neuronas. Hacemos notar que este modelo fue estudiado por los autores en su artículo.
- **Modelo CIFAR-1024**: el modelo está constituido por tres capas «FC» con una disposición de $(1024, 1024, 10)$ neuronas.

Los tres modelos han sido entrenados durante 50 épocas con un «*batch size*» (véase [5]) de 32 imágenes y un «*learning rate*» (véase [5]) de 0.0001. Las capas «FC» han sido inicializadas con pesos extraídos de una distribución normal de media 0 y desviación típica 1. Tras cada época hemos calculado los diagramas de persistencia, el valor de la función de pérdida y la precisión.

5.1.1.1 Diagramas de persistencia de CIFAR-256

En primer lugar, en la Figura 5.1 y en la Figura 5.2 podemos ver los valores de la función de pérdida y precisión del modelo tras cada época. Utilizaremos estos gráficos como guía durante el estudio de los diagramas de persistencia.

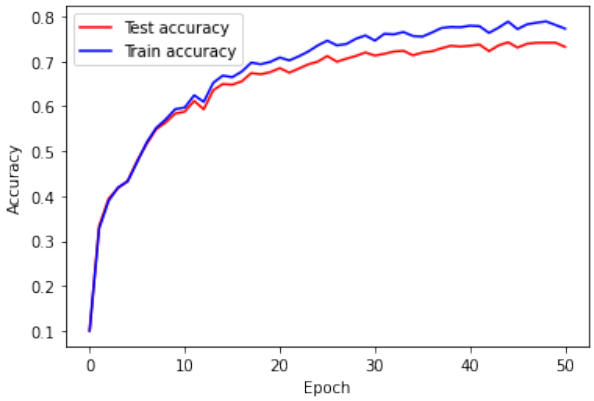

Figura 5.1: Precisión de CIFAR-256 en los conjuntos de entrenamiento y test.

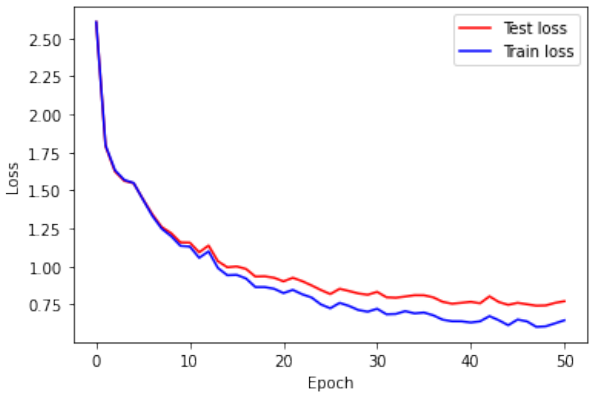

Figura 5.2: Función de pérdida de CIFAR-256 en los conjuntos de entrenamiento y test.

Como ya hemos adelantado, hemos calculado los diagramas de persistencia de cada época, pero no vamos a mostrarlos todos por limitaciones de espacio. En la Figura 5.3, Figura 5.4, Figura 5.5, y en la Figura 5.6 podemos ver algunos diagramas de persistencia de CIFAR-256, donde el primero de ellos se ha calculado a partir del modelo sin entrenar (con pesos aleatorios), y el último se ha calculado sobre el modelo entrenado tras 50 épocas.

Sobre estos diagramas hacemos las siguientes observaciones:

- En líneas generales, el área abarcada por la «nube» de puntos principal (en azul) aumenta a medida que entrenamos el modelo. Esto se debe a que el entrenamiento de la red varía los pesos de tal manera que hay zonas de la red con una distribución de importancias cercanas a 0 (puntos de la zona superior izquierda en la nube), y zonas de la red con una distribución de importancias cercanas a 1 (puntos de la nube cercanos a la diagonal).

- La vertical de puntos que representa las clases de homología de grado 0 (en rojo) aumenta de tamaño. Esto es debido, como hemos mencionado anteriormente, a los cambios que realiza la red en la distribución de los pesos, y en consecuencia de las importancias. En concreto, observamos que a medida que se reducen los pesos (y las importancias) las componentes conexas sobreviven a más filtraciones, y a medida que aumentan, las componentes conexas mueren en las primeras filtraciones.

- La horizontal de puntos que representa las clases de homología que persisten se puebla de clases de grado 1, como consecuencia de la disminución de los pesos y, por tanto, de las importancias.

Aunque los diagramas de persistencia aportan mucha información, dado el elevado número de puntos en ellos no podemos hacernos una idea completa de lo que sucede con la red. Para ello, en la Figura 5.7 podemos ver el número de clases (número de Betti) en función de las épocas de entrenamiento. A la vista de la Figura 5.7, y como veremos en los siguientes experimentos, hacemos la siguiente observación respecto al estudio de la homología persistente en redes neuronales:

Observación 1. El número total de clases de homología se estabiliza cuando la red se vuelve óptima, i.e., ha terminado su aprendizaje.

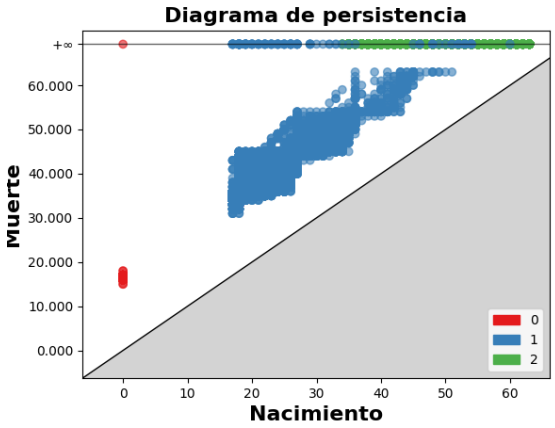

Figura 5.3: Diagrama de persistencia de CIFAR-256 en la época 0.

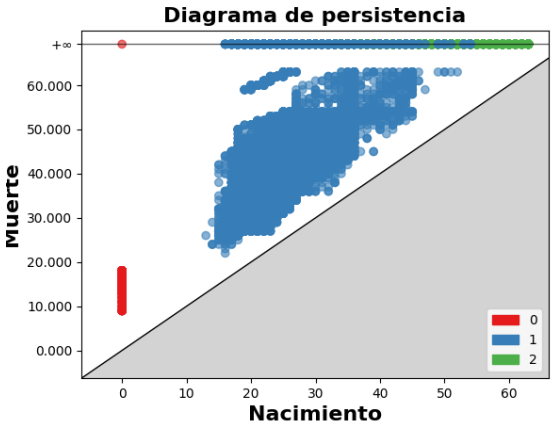

Figura 5.5: Diagrama de persistencia de CIFAR-256 en la época 16.

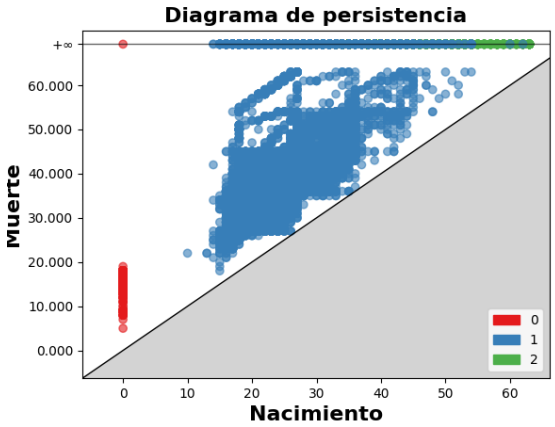

Figura 5.4: Diagrama de persistencia de CIFAR-256 en la época 32.

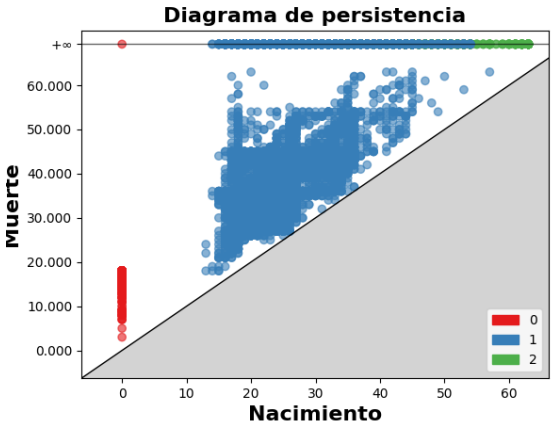

Figura 5.6: Diagrama de persistencia de CIFAR-256 en la época 50.

Nota. De manera intuitiva podemos justificar esta observación como consecuencia de la existencia de al menos un conjunto de pesos óptimo. Es decir, una vez la red alcanza un conjunto óptimo, cada época de entrenamiento adicional modifica muy poco dicho conjunto, y por tanto los números de Betti no acusan tal modificación.

Para apreciar la variación de los números de Betti en función de la época mostramos la Figura 5.8. En ella se ha calculado la tasa de cambio de la siguiente manera:

$$\Delta\beta = \frac{\beta_i - \beta_{i-1}}{\beta_{i-1}}$$

donde β_i es el correspondiente número de Betti en la época i. Ahora, observando la Figura 5.7, la Figura 5.8, y razonando según la Observación 1 concluimos que CIFAR-256 se vuelve óptima hacia la época 40 en su entrenamiento. Acudiendo a la Figura 5.1 y a la Figura 5.2 podemos reafirmar tal conclusión.

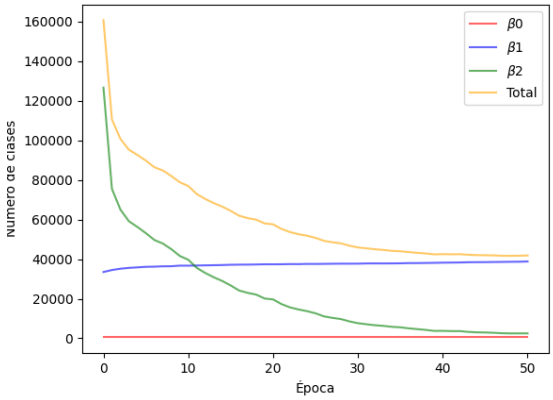

Figura 5.7: Números de Betti asociados a CIFAR-256 en cada época.

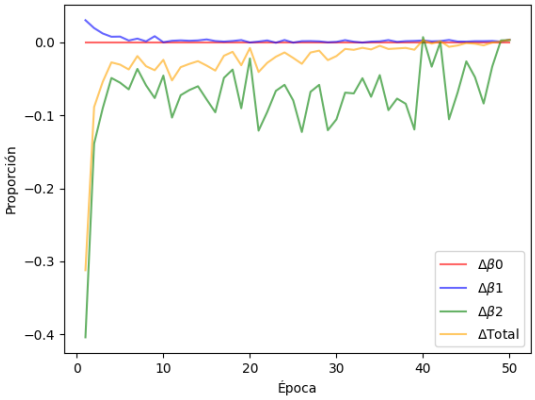

Figura 5.8: Tasa de cambio en los números de Betti asociados a CIFAR-256 en cada época.

Otra cuestión relevante que se deriva de la observación de los diagramas de persistencia es el estudio de las clases de homología que persisten. En concreto, en la Figura 5.9 podemos ver cómo evolucionan en número a lo largo de las épocas, y en la Figura 5.10 podemos ver cómo evoluciona su proporción con el paso de las épocas. En ambos gráficos observamos cómo inicialmente el número/proporción de clases persistentes disminuye, alcanza un mínimo, y vuelve a aumentar. Este comportamiento puede merecer un estudio más profundo, aunque destacamos que al finalizar el entrenamiento, la mayor parte de las clases de homología son persistentes, es decir, en la Figura 5.6 la mayoría de puntos se encuentran sobre la horizontal que marca el infinito.

Finalizamos el estudio de CIFAR-256 estableciendo que el grado de conocimiento adquirido por CIFAR-256 es bajo, pues la mayoría de clases son persistentes, lo que refleja una baja importancia entre muchas familias de neuronas. Esto implica que a la red le basta con un pequeño número de ellas para determinar el resultado, lo que hace que el grado de conocimiento adquirido sea equivalente a la detección de patrones. Destacamos que esta conclusión se deriva del estudio realizado en [12], así como de la experimentación realizada en el presente trabajo.

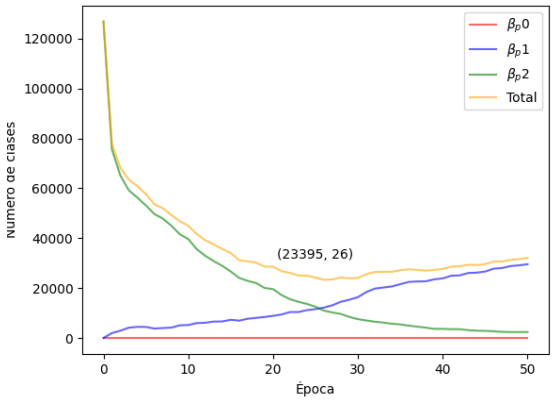

Figura 5.9: Número clases persistentes asociadas a CIFAR-256 en cada época.

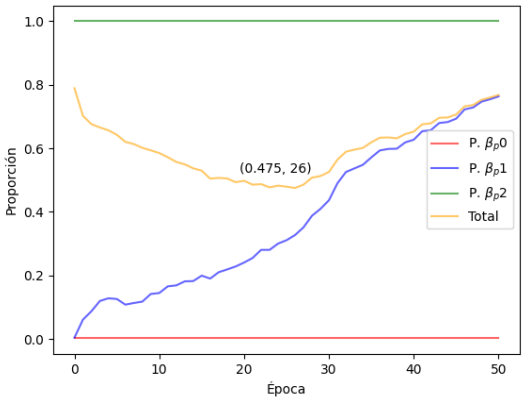

Figura 5.10: Proporción de clases persistentes asociadas a CIFAR-256 en cada época.

5.1.1.2 Diagramas de persistencia de CIFAR-512

Al igual que antes, en la Figura 5.12 y en la Figura 5.11 podemos ver los valores de la función de pérdida y precisión del modelo tras cada época. Utilizaremos estos gráficos como guía durante el estudio de los diagramas de persistencia.

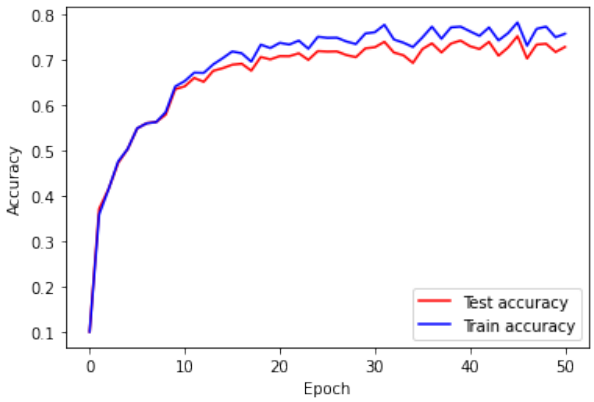

Figura 5.11: Precisión de CIFAR-512 en los conjuntos de entrenamiento y test.

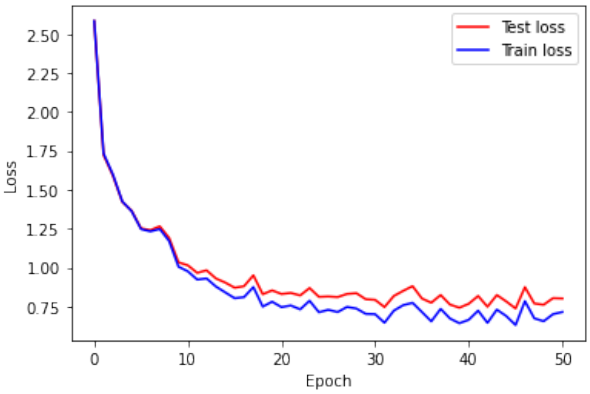

Figura 5.12: Función de pérdida de CIFAR-512 en los conjuntos de entrenamiento y test.

Siguiendo la misma línea que con CIFAR-256, en la Figura 5.13, Figura 5.14, Figura 5.15, y en la Figura 5.16 podemos ver los correspondientes diagramas de persistencia de CIFAR-512.

Sobre estos diagramas podemos hacer las mismas observaciones que con los diagramas de CIFAR-256, con la observación adicional de que el tamaño de la nube principal de puntos aumenta considerablemente respecto al tamaño de la nube principal de CIFAR-256. Este suceso es consecuencia del aumento del tamaño del modelo.

Al igual que con CIFAR-256, resulta conveniente observar la Figura 5.17 y la Figura 5.18, en las que podemos ver la evolución de los números de Betti a través de las épocas de entrenamiento. Si bien ambos gráficos son muy parecidos a los correspondientes de CIFAR-256, hacemos notar que la

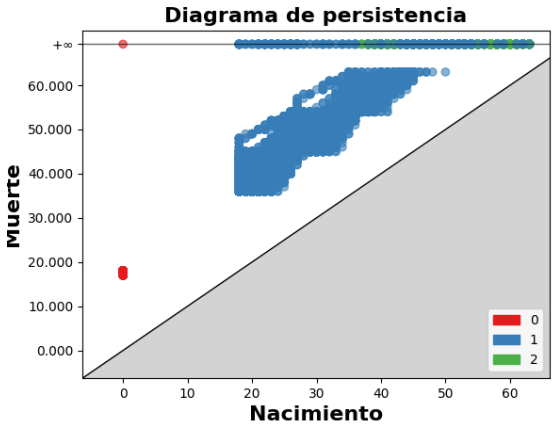

Figura 5.13: Diagrama de persistencia de CIFAR-512 en la época 0.

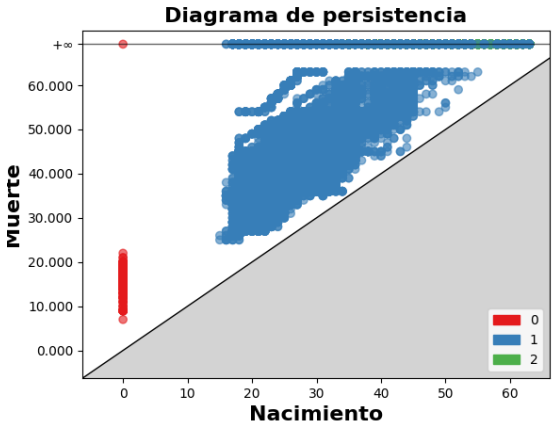

Figura 5.14: Diagrama de persistencia de CIFAR-512 en la época 32.

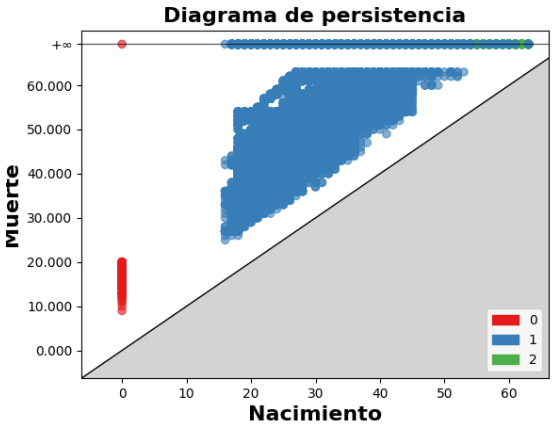

Figura 5.15: Diagrama de persistencia de CIFAR-512 en la época 16.

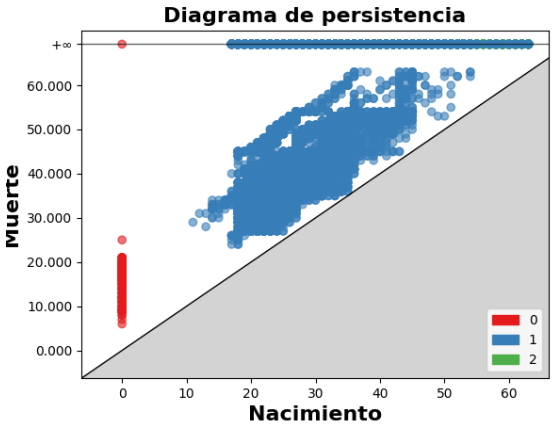

Figura 5.16: Diagrama de persistencia de CIFAR-512 en la época 50.

estabilización del número de clases de homología se produce hacia la época 30 de entrenamiento, lo que, en virtud de la Observación 1, nos hace concluir que el modelo resulta óptimo con 30 épocas de entrenamiento. Una vez más, podemos reafirmar tal conclusión acudiendo a la Figura 5.11 y a la Figura 5.12.

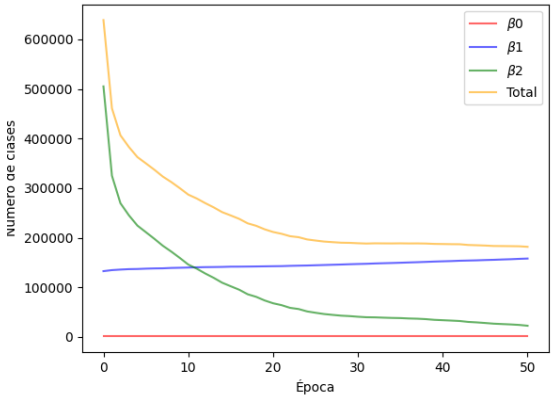

Figura 5.17: Números de Betti asociados a CIFAR-512 en cada época.

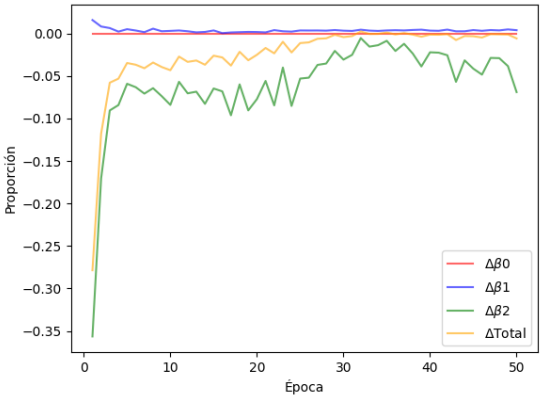

Figura 5.18: Tasa de cambio en los números de Betti asociados a CIFAR-512 en cada época.

En lo referente a las clases de homología que persisten, en la Figura 5.19 y en la Figura 5.20 podemos ver los gráficos correspondientes. En ambos gráficos apreciamos un comportamiento similar al de CIFAR-256, aunque existen algunas diferencias: el mínimo en el número y proporción de clases persistentes se produce antes que en CIFAR-256, y la proporción de clases persistentes al finalizar el entrenamiento es mayor que en CIFAR-256.

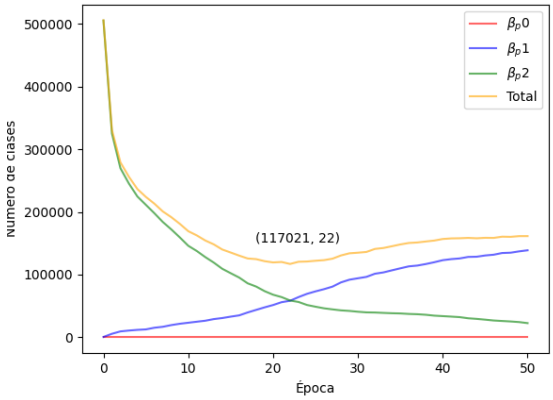

Figura 5.19: Número clases persistentes asociadas a CIFAR-512 en cada época.

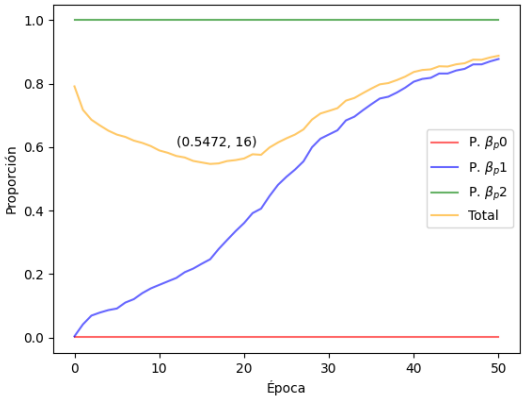

Figura 5.20: Proporción de clases persistentes asociadas a CIFAR-512 en cada época.

Al igual que con CIFAR-256, establecemos que el grado de conocimiento adquirido por CIFAR-512 es bajo, y menor que el adquirido por CIFAR-256, pues la mayoría de clases son persistentes, y la proporción de clases persistentes es mayor que la de CIFAR-256. Este suceso refleja un mayor

excedente de familias de neuronas para la correcta resolución del problema.

5.1.1.3 Diagramas de persistencia de CIFAR-1024

Siguiendo la línea anterior, en la Figura 5.21 y en la Figura 5.22 podemos ver los valores de la función de pérdida y precisión del modelo tras cada época. Estos gráficos nos servirán de guía durante el estudio de los diagramas de persistencia.

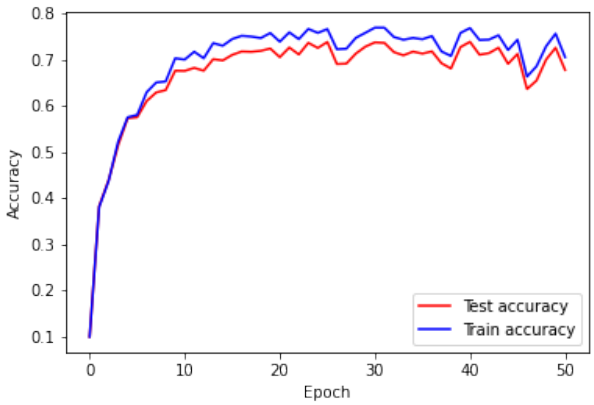

Figura 5.21: Precisión de CIFAR-1024 en los conjuntos de entrenamiento y test.

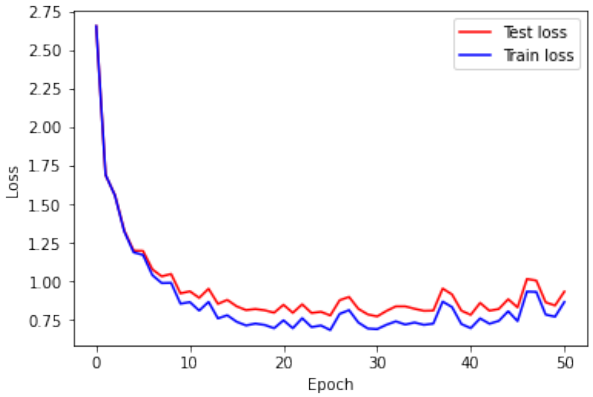

Figura 5.22: Función de pérdida de CIFAR-1024 en los conjuntos de entrenamiento y test.

Ahora, en la Figura 5.23, Figura 5.24, Figura 5.25, y en la Figura 5.26 podemos ver los diagramas de persistencia asociados a CIFAR-1024.

Observamos que estos diagramas presentan claras diferencias respecto a los diagramas de persistencia anteriores, a saber:

- La nube principal de puntos se mantiene más cohesionada que en los anteriores modelos, y a medida que se entrena el modelo se va dispersando y aumentando su tamaño.
- La vertical que representa las clases de homología de grado 0 (en rojo) acaba teniendo una longitud mayor que en lo modelos anteriores, e incluso se puede apreciar que se produce una separación al finalizar el entrenamiento. Esto se debe a que la red está aislando (haciendo que sus importancias circundantes sean próximas a 0) ciertas familias de neuronas, causando que las clases que las representan vivan durante más filtraciones, e integrando (aumentando alguna de sus importancias circundantes) otras, haciendo que mueran en las primeras filtraciones.

Las observaciones previas, aunque interesantes, no nos permiten llegar a conclusiones por sí solas. Para tener una idea más completa debemos acudir a la Figura 5.27 y la Figura 5.28, en las que podemos ver la evolución de los números de Betti a través de las épocas de entrenamiento. En estos gráficos podemos observar dos particularidades respecto de los correspondientes gráficos de los modelos anteriores: en la Figura 5.27 se aprecia una momentánea estabilización en el número de clases de homología sobre la época 20, y en la Figura 5.28 se aprecia un comportamiento oscilatorio en la tasa de cambio de β_2 al finalizar el entrenamiento. La segunda particularidad queda explicada por el reducido valor de β_2 al finalizar el entrenamiento, haciendo que un ligero cambio represente una mayor proporción.

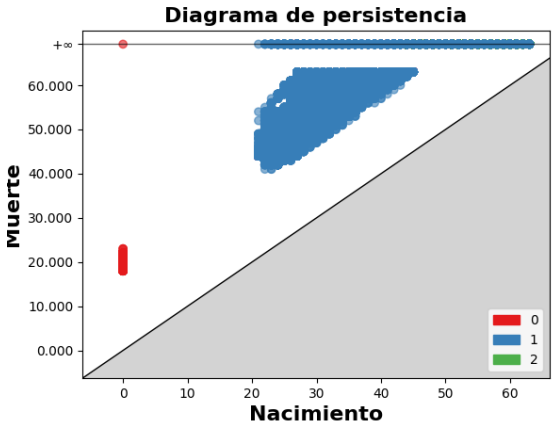

Figura 5.23: Diagrama de persistencia de CIFAR-1024 en la época 0.

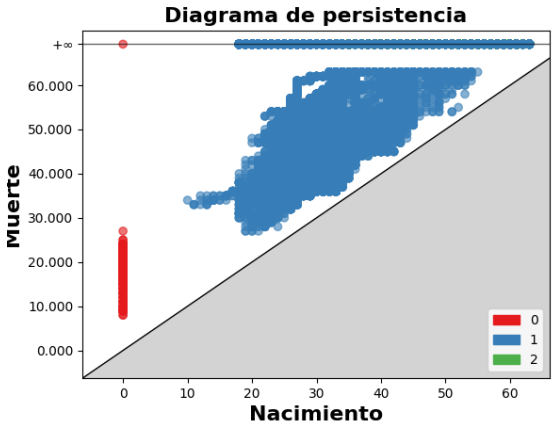

Figura 5.24: Diagrama de persistencia de CIFAR-1024 en la época 32.

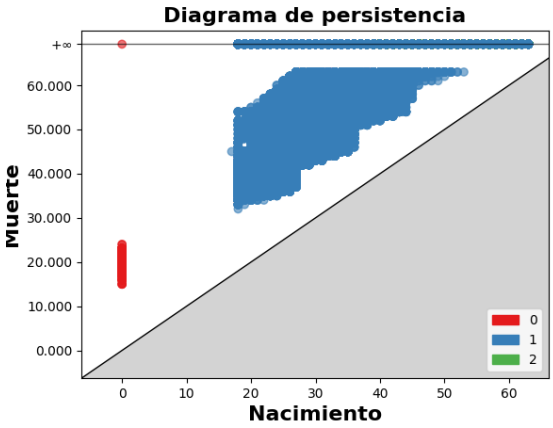

Figura 5.25: Diagrama de persistencia de CIFAR-1024 en la época 16.

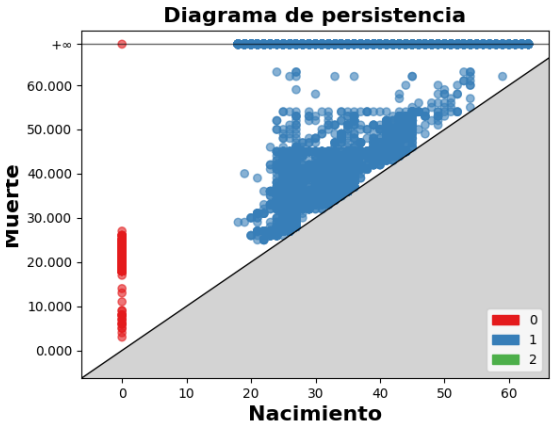

Figura 5.26: Diagrama de persistencia de CIFAR-1024 en la época 50.

En cualquier caso, hacemos notar que la estabilización (final) del número de clases de homología se produce un poco antes de la época 40 de entrenamiento, lo que, en virtud de la Observación 1, nos hace concluir que el modelo resulta óptimo con 40 épocas de entrenamiento. Una vez más, podemos reafirmar tal conclusión acudiendo a la Figura 5.21 y a la Figura 5.22.

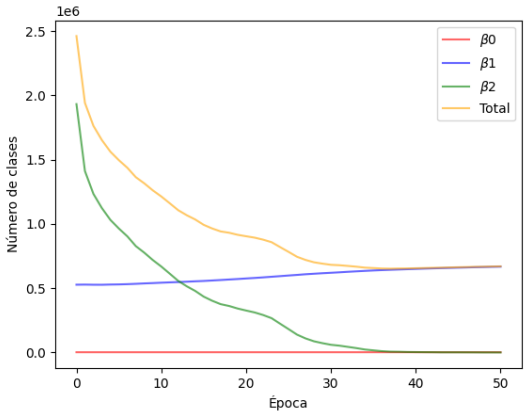

Figura 5.27: Números de Betti asociados a CIFAR-1024 en cada época.

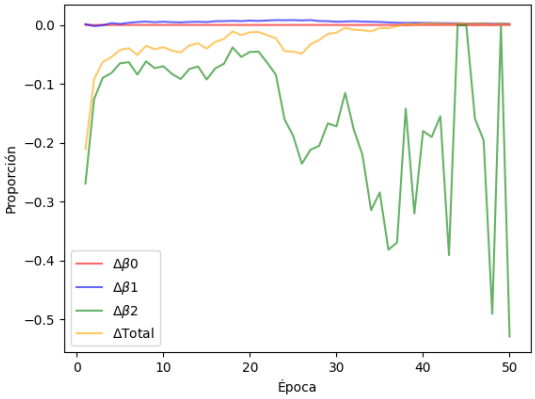

Figura 5.28: Tasa de cambio en los números de Betti asociados a CIFAR-1024 en cada época.

En lo referente a las clases de homología que persisten, en la Figura 5.29 y en la Figura 5.30 podemos ver los gráficos correspondientes. Como consecuencia directa de la primera particularidad que podíamos observar en la Figura 5.27, observamos que se produce un máximo local en la Figura 5.29. Ahora, en la Figura 5.30 podemos observar un comportamiento similar al de los anteriores modelos, con la salvedad de que el número de clases persistentes es aún mayor en este caso.

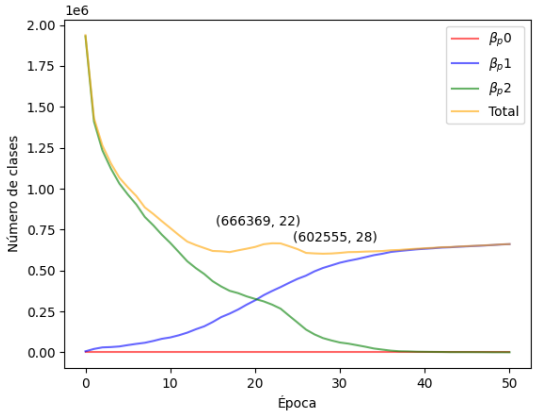

Figura 5.29: Número clases persistentes asociadas a CIFAR-1024 en cada época.

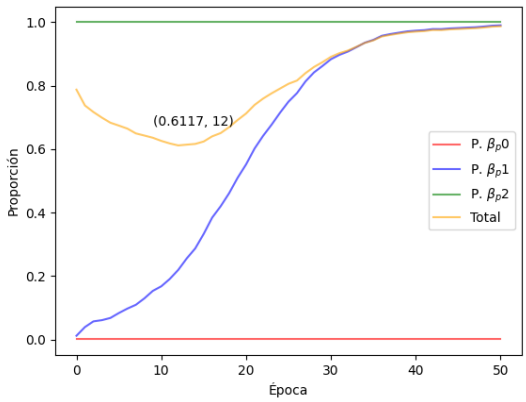

Figura 5.30: Proporción de clases persistentes asociadas a CIFAR-1024 en cada época.

Como veníamos razonando anteriormente, establecemos que el grado de conocimiento adquirido por CIFAR-1024 es muy bajo, e inferior que el adquirido por CIFAR-256 y CIFAR-512, pues la proporción de clases persistentes en este caso es mayor que la de los casos anteriores. Este hecho

denota un mayor excedente de familias de neuronas en la red.

5.1.2 Modelos MNIST

Se han entrenado tres modelos de visión por computador sobre el dataset MNIST (7000 imágenes de 28x28 píxeles de dígitos manuscritos). La tarea a realizar por los modelos consiste en clasificar las imágenes, en blanco y negro, en 10 categorías distintas, según el dígito representado en cada imagen. Los tres modelos están compuestos únicamente por capas completamente conectadas (FC por sus siglas en inglés) que hemos variado de tamaño, en concreto tenemos los modelos:

- **Modelo MNIST-150**: el modelo está constituido por tres capas «FC» con una disposición de $(150, 50, 10)$ neuronas.
- **Modelo MNIST-300**: el modelo está constituido por tres capas «FC» con una disposición de $(300, 100, 10)$ neuronas. Hacemos notar que este modelo fue estudiado en [12].
- **Modelo MNIST-600**: el modelo está constituido por tres capas «FC» con una disposición de $(600, 200, 10)$ neuronas.

Los tres modelos han sido entrenados durante 50 épocas con un «*batch size*» (véase [5]) de 64 imágenes y un «*learning rate*» (véase [5]) de 0.001. Las capas «FC» han sido inicializadas con pesos extraídos de una distribución $U(-\sqrt{\frac{6}{n_i+n_j}}, \sqrt{\frac{6}{n_i+n_j}})$, donde n_i y n_j son el número de entradas y salidas de la capa respectivamente. Tras cada época hemos calculado los diagramas de persistencia, el valor de la función de pérdida y la precisión.

5.1.2.1 Diagramas de persistencia de MNIST-150

Al igual que con los modelos CIFAR, en la Figura 5.31 y en la Figura 5.32 podemos ver los valores de la función de pérdida y precisión del modelo tras cada época. En estos gráficos destacamos la rápida convergencia del modelo a una solución óptima. Esta diferencia respecto a los modelos CIFAR es común a los tres modelos MNIST estudiados.

En la Figura 5.33, la Figura 5.34, la Figura 5.35, y en la Figura 5.36 podemos ver los diagramas de persistencia asociados a MNIST-150.

Sobre los anteriores diagramas hacemos las siguientes observaciones:

- El entrenamiento de este modelo modifica muy superficialmente los diagramas de persistencia, cuyas únicas diferencias apreciables las podemos encontrar en la zona superior derecha de los diagramas.
- Apenas aparecen clases de homología de grado 1 persistentes.

Para completar los diagramas de persistencia mostramos la Figura 5.37 y la Figura 5.38. En ellas podemos observar, al igual que con los modelos CIFAR, que el número de clases de homología se estabiliza en cierto punto del entrenamiento. Ahora bien, en la Figura 5.37 observamos que β_2 no se reduce drásticamente, como sí sucedía con los modelos CIFAR, y en la Figura 5.38 podemos observar que las tasas de cambio no superan el 8 %.

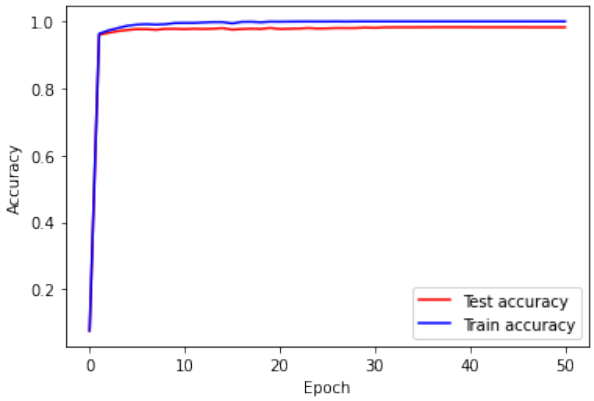

Figura 5.31: Precisión de MNIST-150 en los conjuntos de entrenamiento y test.

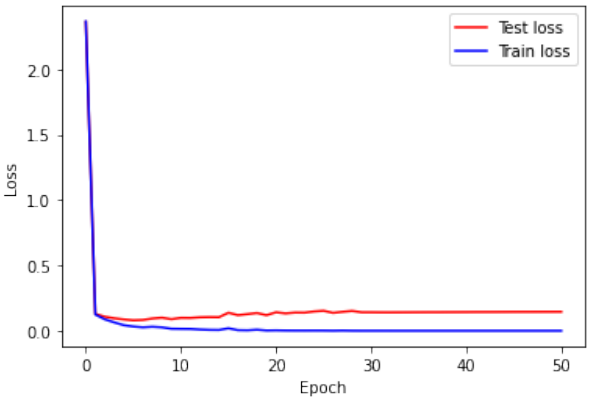

Figura 5.32: Función de pérdida de MNIST-150 en los conjuntos de entrenamiento y test.

Ahora, aplicando la Observación 1 concluimos que este modelo se hace óptimo con unas 30 épocas de entrenamiento. A diferencia de los modelos CIFAR, dada la rápida convergencia de este modelo y la inapreciable diferencia en la precisión y pérdida a lo largo de las épocas (estamos analizando diferencias de precisión de entre el 1 % y el 0.01 %), no podemos reafirmar de manera clara esta conclusión acudiendo a la Figura 5.31 y a la Figura 5.32.

En cuanto a las clases de homología que persisten, en la Figura 5.39 y en la Figura 5.40 podemos ver los gráficos correspondientes. Observamos que la mayoría de las clases de homología persisten, y que casi todas ellas son de grado 2, a diferencia de lo que sucedía con los modelos CIFAR.

En contraposición a la línea argumentativa previa, podemos establecer que el grado de conocimiento adquirido por MNIST-150 es alto, a diferencia del adquirido por los modelos CIFAR, pues el número/proporción de clases persistentes es ínfimo. Nótese que dada la naturaleza de la red, las clases de homología de grado 2 (en verde) siempre son persistentes, por lo que no son relevantes en este estudio.

5.1.2.2 Diagramas de persistencia de MNIST-300

Nuevamente, en la Figura 5.41 y en la Figura 5.42 podemos ver los valores de la función de pérdida y precisión del modelo tras cada época. Estos gráficos no presentan diferencias apreciables respecto a los correspondientes gráficos de MNIST-150.

En la Figura 5.43, la Figura 5.44, la Figura 5.45, y en la Figura 5.46 podemos ver los diagramas de persistencia asociados a MNIST-300.

Sobre los diagramas de persistencia de MNIST-300 observamos:

- La nube principal de puntos (en azul) se ve ligeramente modificada con las épocas de entrenamiento. Principalmente observamos cambios de tamaño en la zona central de la nube.
- A diferencia de lo sucedido con MNIST-150, observamos evidentes cambios en la distribución de los puntos que representan clases de homología de grado 0 (en rojo). En concreto, podemos observar una división de los puntos en tres grupos: el grupo superior, el central y el inferior. La

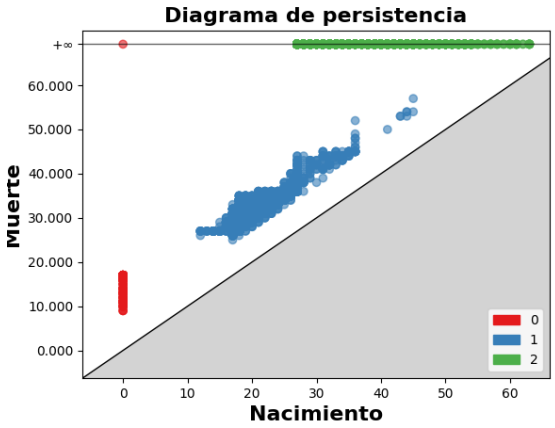

Figura 5.33: Diagrama de persistencia de MNIST-150 en la época 0.

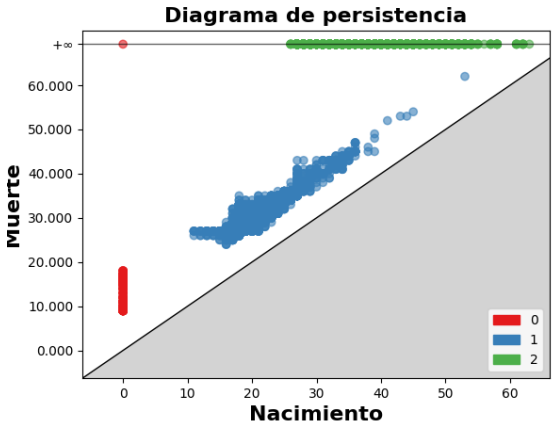

Figura 5.34: Diagrama de persistencia de MNIST-150 en la época 32.

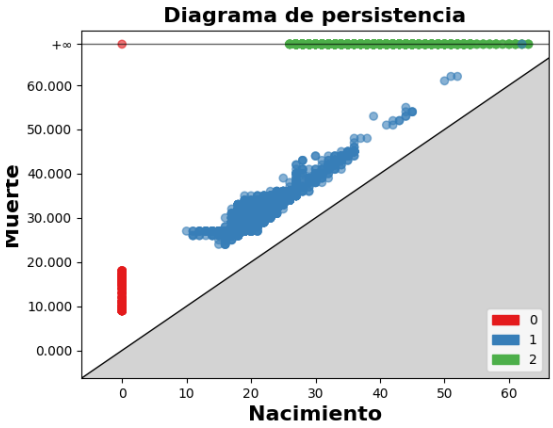

Figura 5.35: Diagrama de persistencia de MNIST-150 en la época 16.

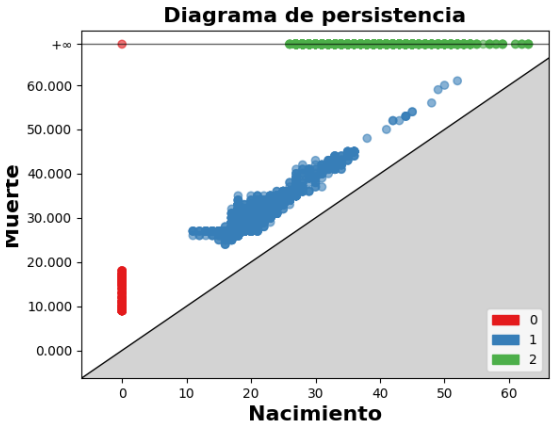

Figura 5.36: Diagrama de persistencia de MNIST-150 en la época 50.

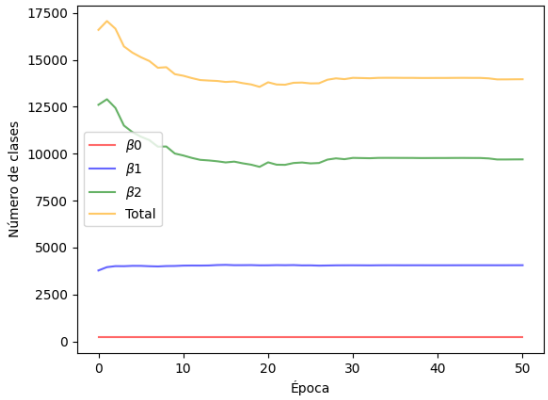

Figura 5.37: Números de Betti asociados a MNIST-150 en cada época.

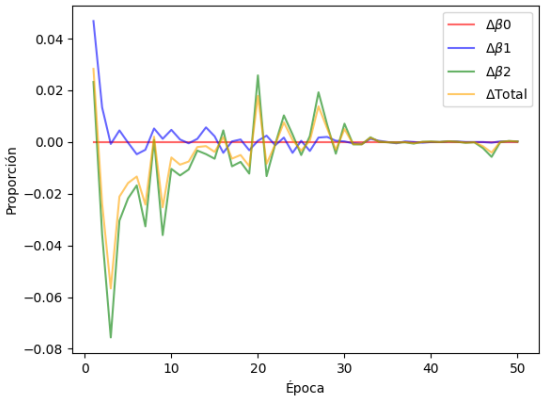

Figura 5.38: Tasa de cambio en los números de Betti asociados a MNIST-150 en cada época.

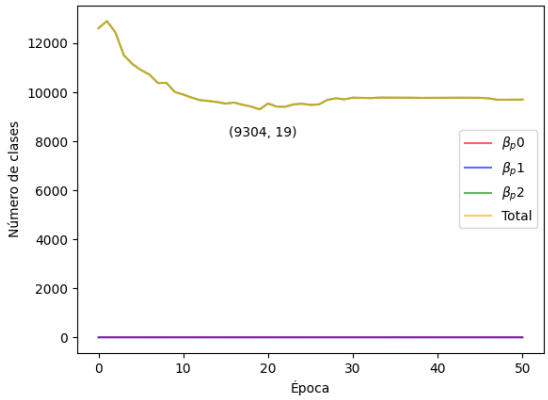

Figura 5.39: Número clases persistentes asociadas a MNIST-150 en cada época.

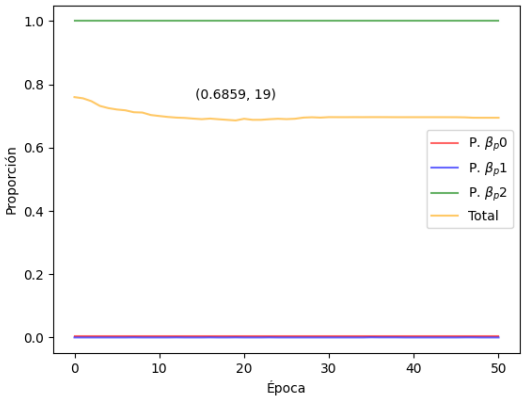

Figura 5.40: Proporción de clases persistentes asociadas a MNIST-150 en cada época.

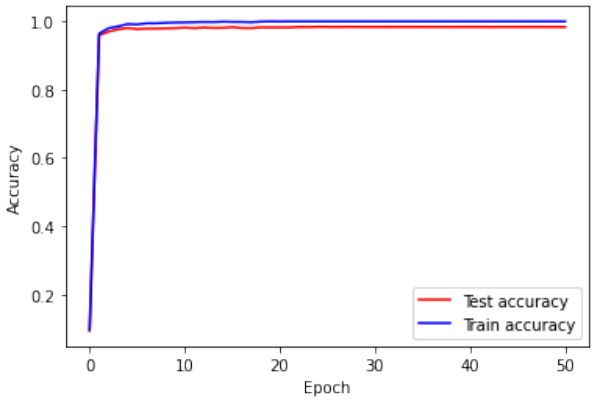

Figura 5.41: Precisión de MNIST-300 en los conjuntos de entrenamiento y test.

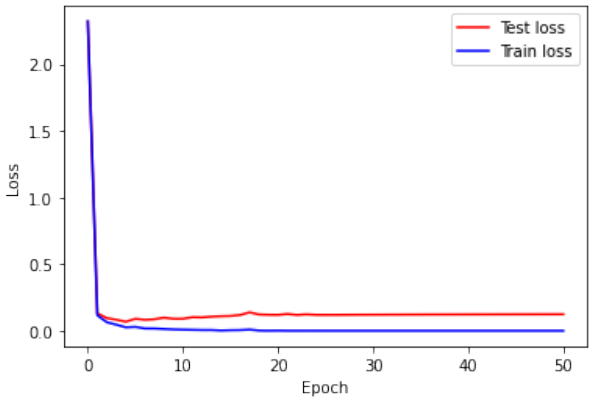

Figura 5.42: Función de pérdida de MNIST-300 en los conjuntos de entrenamiento y test.

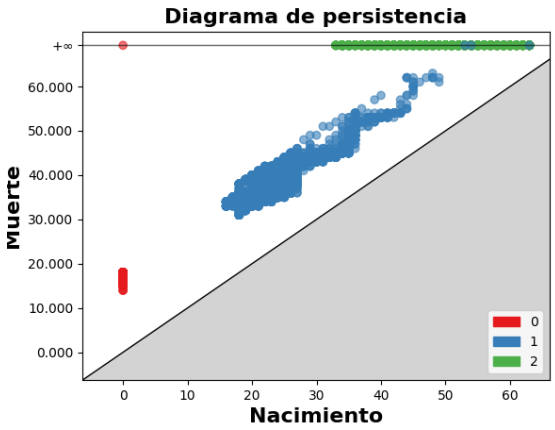

Figura 5.43: Diagrama de persistencia de MNIST-300 en la época 0.

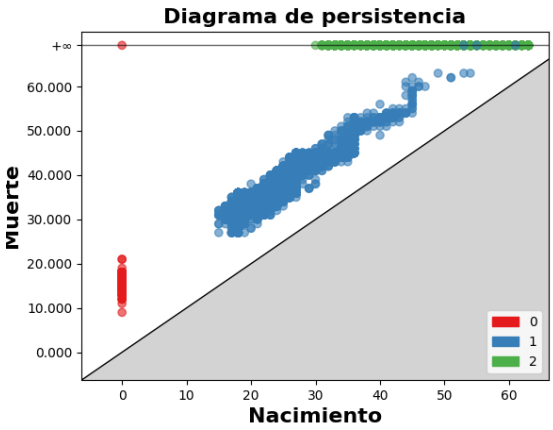

Figura 5.45: Diagrama de persistencia de MNIST-300 en la época 16.

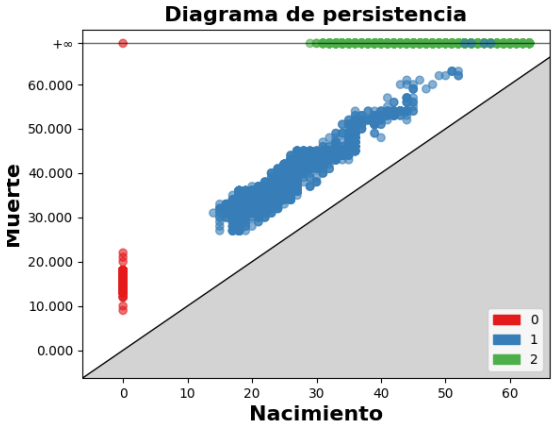

Figura 5.44: Diagrama de persistencia de MNIST-300 en la época 32.

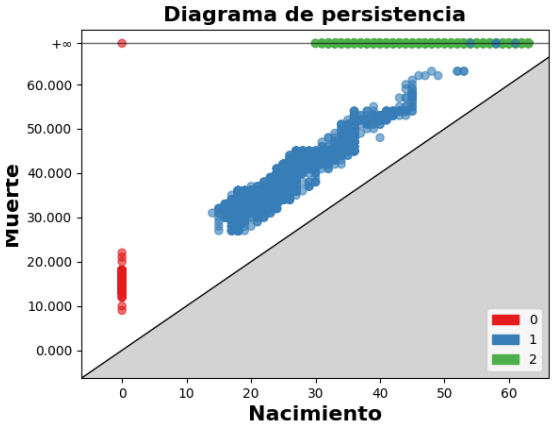

Figura 5.46: Diagrama de persistencia de MNIST-300 en la época 50.

aparición de los grupos superior e inferior implica una redistribución de la importancia entre las familias de neuronas aislando algunas y comunicando otras.

- Aparición de algunas clases de homología de grado 1 persistentes.

Acudiendo a la Figura 5.47 y a la Figura 5.48 podemos observar el mismo comportamiento que en MNIST-150, con la particularidad de la oscilación en la tasa de cambio de las clases de homología que podemos ver en la Figura 5.48. Nótese que tales oscilaciones no alcanzan el 2 %, por lo que resultan despreciables.

Una vez más, aplicando la Observación 1, llegamos a la conclusión de que el modelo se hace óptimo hacia la época 25 de entrenamiento. Como ya hemos visto anteriormente, en este caso no podemos confirmar de manera clara esta conclusión acudiendo a la Figura 5.41 y a la Figura 5.42.

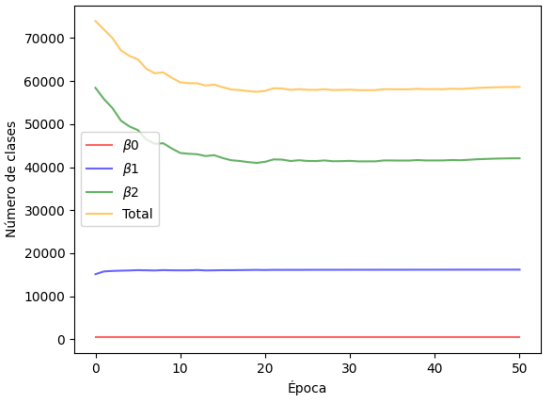

Figura 5.47: Números de Betti asociados a MNIST-300 en cada época.

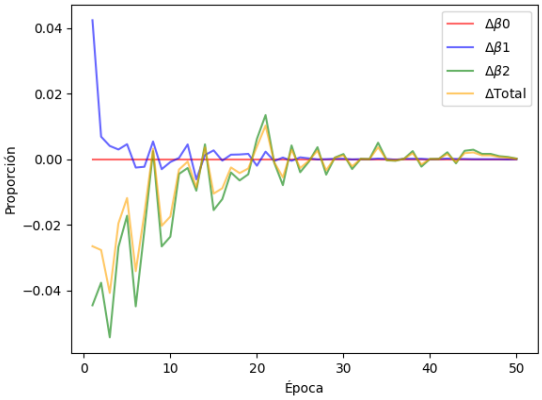

Figura 5.48: Tasa de cambio en los números de Betti asociados a MNIST-300 en cada época.

Para el estudio de las clases de homología que persisten, en la Figura 5.49 y en la Figura 5.50 podemos ver los gráficos correspondientes. En esta ocasión podemos observar el mismo comportamiento que en MNIST-150, con la salvedad de un pequeño incremento en la proporción de clases persistentes.

Tal y como hemos argumentado con MNIST-150, podemos establecer que el grado de conocimiento adquirido por MNIST-300 es alto, pero menor que el adquirido por MNIST-150, pues la proporción de clases persistentes es mayor.

5.1.2.3 Diagramas de persistencia de MNIST-600

Finalmente, en la Figura 5.51 y en la Figura 5.52 podemos ver los valores de la función de pérdida y precisión del modelo tras cada época. Estos gráficos no presentan diferencias apreciables respecto a los correspondientes gráficos de MNIST-150 y MNIST-300.

En la Figura 5.53, la Figura 5.54, la Figura 5.55, y en la Figura 5.56 podemos ver los diagramas de persistencia asociados a MNIST-600.

Sobre los diagramas de persistencia de MNIST-600 observamos:

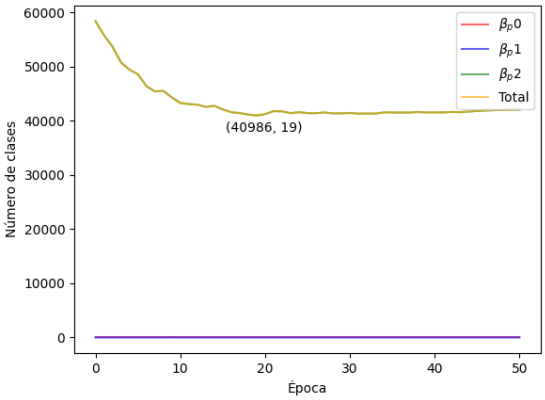

Figura 5.49: Número clases persistentes asociadas a MNIST-300 en cada época.

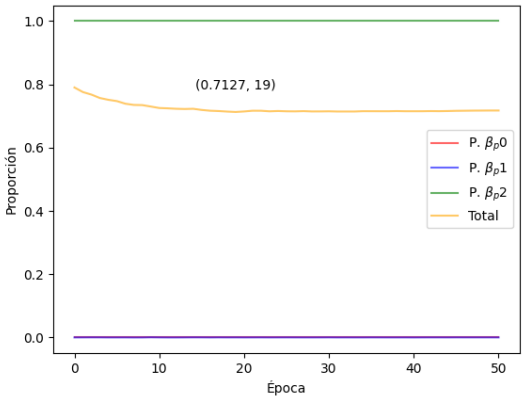

Figura 5.50: Proporción de clases persistentes asociadas a MNIST-300 en cada época.

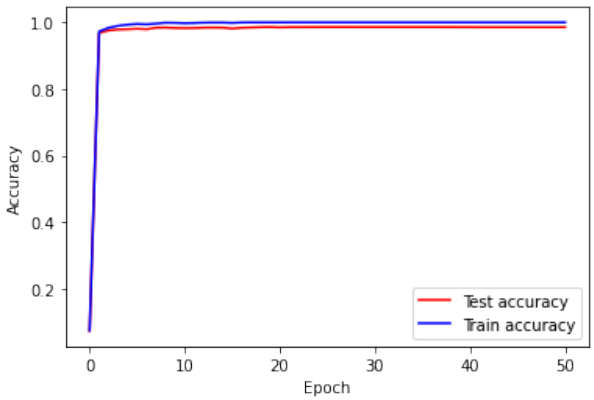

Figura 5.51: Precisión de MNIST-600 en los conjuntos de entrenamiento y test.

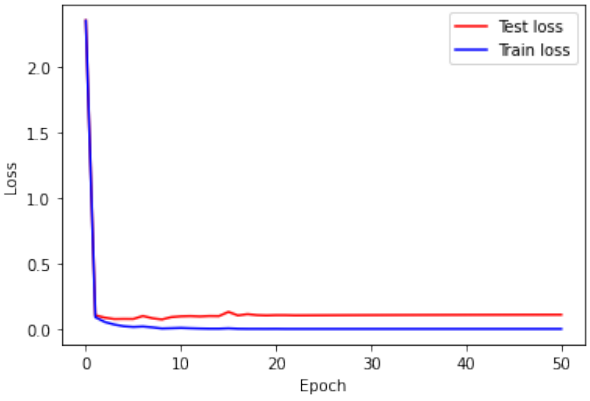

Figura 5.52: Función de pérdida de MNIST-600 en los conjuntos de entrenamiento y test.

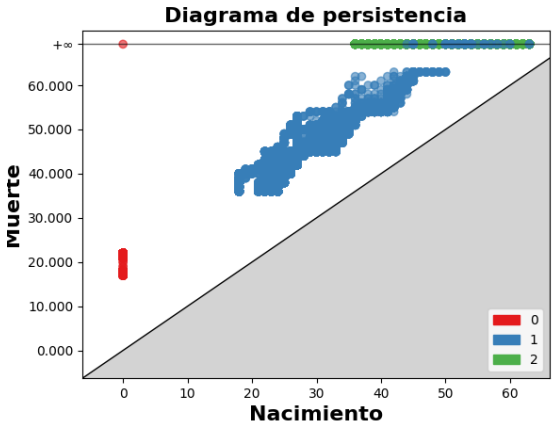

Figura 5.53: Diagrama de persistencia de MNIST-600 en la época 0.

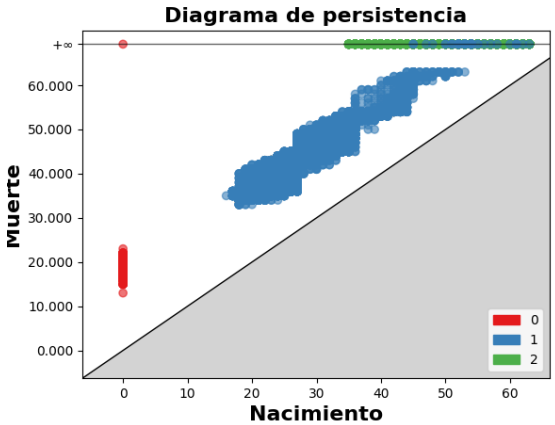

Figura 5.54: Diagrama de persistencia de MNIST-600 en la época 32.

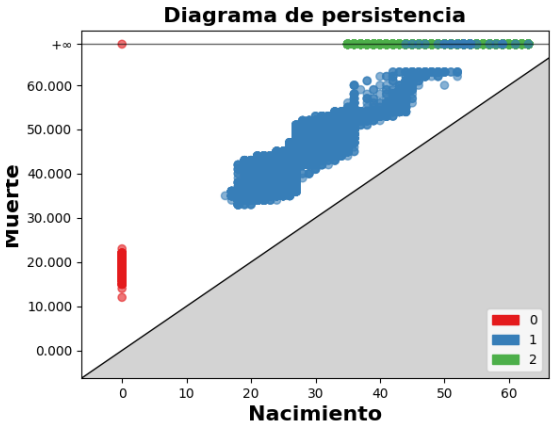

Figura 5.55: Diagrama de persistencia de MNIST-600 en la época 16.

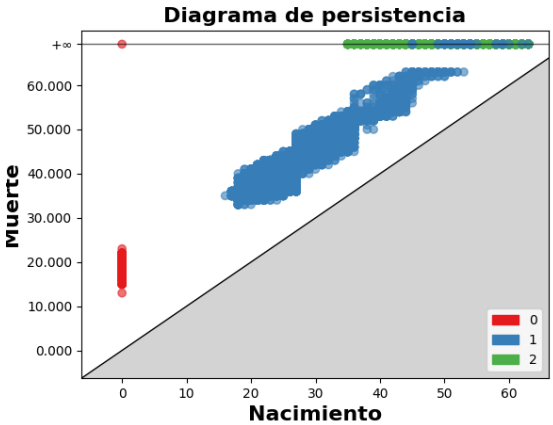

Figura 5.56: Diagrama de persistencia de MNIST-600 en la época 50.

- La nube principal de puntos (en azul) se ve ligeramente modificada con las épocas de entrenamiento. Principalmente observamos cambios de tamaño en la zona central de la nube.
- Ligeros cambios en la distribución de los puntos que representan clases de homología de grado 0 (en rojo) respecto de MNIST-300. En concreto, podemos observar un intento de división de los puntos en los tres grupos que veíamos en MNIST-300. En este caso se quedan en un grupo central ligeramente distribuido, pudiendo apreciar un punto en la zona inferior, diferenciado del resto del grupo.
- Aumento de las clases de homología de grado 1 persistentes respecto de MNIST-300.

Consultando la Figura 5.57 y la Figura 5.58 podemos observar el mismo comportamiento que en MNIST-300, aunque en esta ocasión en la Figura 5.58 ha desaparecido el movimiento oscilatorio que veíamos en la Figura 5.48.

Haciendo uso de la Observación 1, concluimos que el modelo se hace óptimo hacia la época 20 de entrenamiento. Como ya hemos visto anteriormente, en este caso no podemos confirmar de manera clara esta conclusión acudiendo a la Figura 5.51 y a la Figura 5.52.

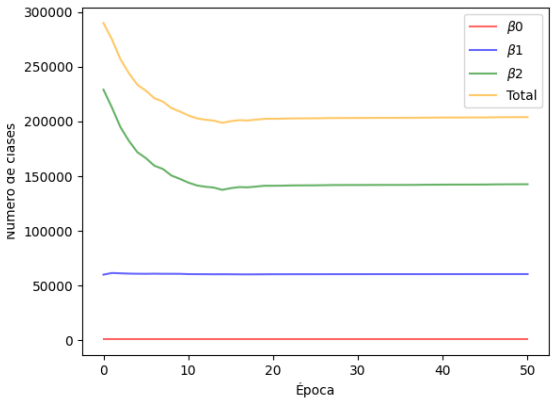

Figura 5.57: Números de Betti asociados a MNIST-600 en cada época.

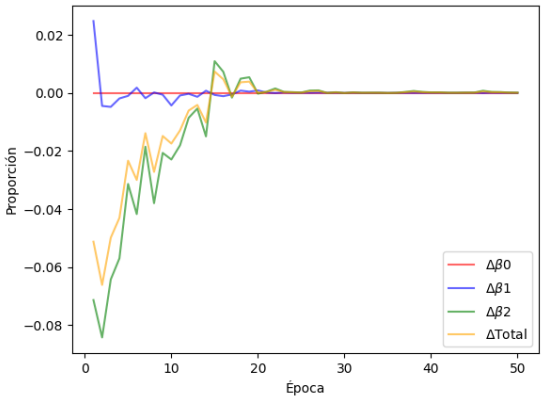

Figura 5.58: Tasa de cambio en los números de Betti asociados a MNIST-600 en cada época.

Para el estudio de las clases de homología que persisten, en la Figura 5.59 y en la Figura 5.60 podemos ver los gráficos correspondientes. En esta ocasión podemos observar el mismo comportamiento que en MNIST-150 y MNIST-300, con la salvedad de un pequeño decremento en la proporción de clases persistentes.

Como hemos estado argumentando hasta ahora, podemos establecer que el grado de conocimiento adquirido por MNIST-600 es alto, y ligeramente superior que el adquirido por MNIST-300, pues la proporción de clases persistentes es menor. Si comparamos MNIST-600 con MNIST-150, podemos establecer que el grado de conocimiento adquirido es similar en ambos modelos, aunque es ligeramente superior en MNIST-150.

En conclusión, observamos que por medio de los diagramas de persistencia y los números de Betti asociados a los ejemplos estudiados, hemos podido estimar el número óptimo de épocas de entrenamiento de cada red neuronal y su grado de conocimiento adquirido. Aunque para extrapolar

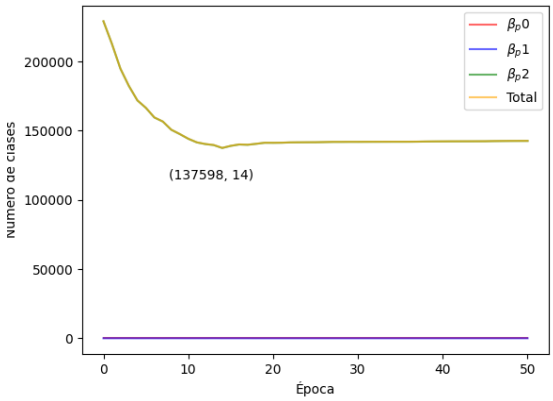

Figura 5.59: Número clases persistentes asociadas a MNIST-600 en cada época.

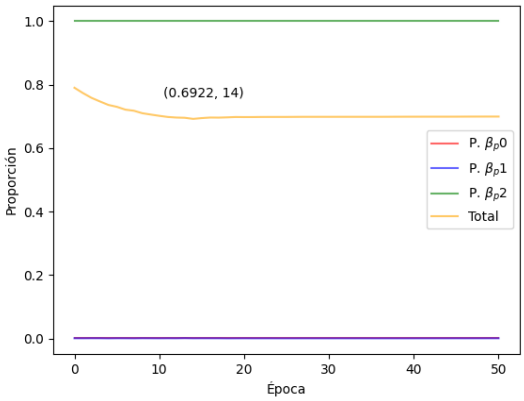

Figura 5.60: Proporción de clases persistentes asociadas a MNIST-600 en cada época.

estas conclusiones a otros modelos más generales es necesario un estudio más profundo, estos experimentos ponen de manifiesto la importancia de la arquitectura de una red neuronal frente al número de épocas de entrenamiento.

Capítulo 6
Seguimiento y control

En este capítulo vamos a exponer las tareas de seguimiento y control del proyecto. Hacemos notar que, dada la naturaleza del proyecto, las tareas de seguimiento y control han consistido en una serie de reuniones con los tutores académicos, que a su vez han ejercido de clientes. Además de las reuniones, hemos mantenido un control aproximado del tiempo empleado en cada fase del proyecto, a fin de analizar las desviaciones que se han producido respecto del tiempo planificado.

6.1 Reuniones

En lo referente a las reuniones, no se acordó ningún plan de reuniones ni contacto periódico. Sin embargo, hemos mantenido un contacto muy cercano debido a la concesión de la beca de iniciación a la investigación, de manera que hemos podido celebrar reuniones de seguimiento y orientación cuando ha sido necesario. En la Figura 6.1 podemos ver un calendario en el que hemos marcado las reuniones celebradas, así como los hitos más importantes del proyecto. A continuación detallamos las reuniones mantenidas con su temática:

- 8 de febrero: análisis de viabilidad y planificación.
- 6 de marzo: análisis del software y diseño.
- 29 de marzo: revisión del desarrollo e introducción a la experimentación.
- 16 de mayo: revisión y control de la memoria. Ajuste de contenidos.
- 6 de junio: revisión y control de la memoria. Correcciones de la memoria.
- 27 de junio: revisión y control de la memoria. Correcciones previas al cierre de la memoria.

6.2 Desviaciones

Tal y como podemos apreciar en la Figura 6.1, se han producido varias desviaciones en cuanto a los hitos estimados (rayados). Estas desviaciones tienen su origen principalmente en la propia naturaleza del proyecto, lo que ha causado que el TFG haya requerido una mayor cantidad de tiempo en sus fase de experimentación. En la Tabla 6.1 podemos ver un desglose de las desviaciones temporales del proyecto.

Figura 6.1: Calendario del proyecto con las reuniones, los hitos reales y los estimados.

Desviaciones del proyecto					
Nombre	Inicio	Cierre	Inicio real	Cierre real	Desviación
1. Planificación	S5	S7	S5	S7	$0\,\%$
2. Análisis y diseño	S8	S11	S8	S10	$-25\,\%$
3. Desarrollo	S12	S17	S10	S15	$0\,\%$
4. Experimentación	S18	S23	S13	S21	$+100\,\%$
5. Memoria	S5	S27	S6	S27	$-5\,\%$
5. Seguimiento y control	S5	S27	S6	S27	$-5\,\%$
Total	S5	S27	S5	S27	$0\,\%$

Tabla 6.1: Balance temporal del proyecto.

Conclusiones

Finalizamos el presente trabajo habiendo aportado significado al uso de la homología persistente para estudiar redes neuronales, así como habiendo desarrollado un software escalable que facilita su aplicación. Desde el punto de vista científico, este trabajo nos ha servido para explorar la aplicación las herramientas estudiadas en [8]. Hacemos notar que mediante el estudio realizado en el trabajo de matemáticas y en el presente trabajo, podemos apreciar el potencial de la herramienta cuyo desarrollo completo dejamos como trabajo futuro.

En retrospectiva, en este trabajo he tenido la oportunidad de poner en práctica algunos de los conocimientos adquiridos en el grado sobre un proyecto propio. En lo técnico, he podido trabajar sobre las disciplinas del cálculo científico, haciendo uso de nociones de optimización y paralelización, y la inteligencia artificial, entrenando y estudiando redes neuronales. En general, también he tenido la ocasión de aplicar conocimientos transversales como el diseño y desarrollo «limpio» de software, y la gestión de proyectos. Considero que las competencias que he adquirido en el grado, y que he podido demostrar con este trabajo, resultarán fundamentales para mi desarrollo profesional futuro.

A título personal, quiero expresar mi más sincera gratitud con el departamento de Matemáticas y Computación por brindarme la oportunidad de desarrollar una beca de iniciación a la investigación sobre este tema, y en particular con mis tutores José Luis y Julio por ofrecerme su guía durante este proyecto. He disfrutado verdaderamente con la labor de investigación realizada en este trabajo y espero tener la ocasión de repetir esta experiencia en el futuro.

Bibliografía

[1] *Modelio - an open source uml / bpmn modeling tool*, 2011. https://www.modelio.org/.

[2] Almeida, Francisco y Domingo Giménez: *Introducción a la programación paralela*. Paraninfo Cengage Learning, Madrid, 2008, ISBN 978-84-9732-674-2.

[3] Bauer, Ulrich: *Ripser: efficient computation of Vietoris-Rips persistence barcodes*. J. Appl. Comput. Topol., 5(3):391–423, 2021, ISSN 2367-1726. https://doi.org/10.1007/s41468-021-00071-5.

[4] Divasón, Jose: *Notas del curso de sistemas distribuidos*, 2022.

[5] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville: *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[6] Kottwitz, Stefan: *LaTeX Cookbook*. Packt open source. Packt Publishing, 2015, ISBN 9781784395148. https://books.google.es/books?id=j9ijjgEACAAJ.

[7] Morozov, Dmitriy: *Dionysus 2 - library for computing persistent homology*, May 2021. https://mrzv.org/software/dionysus2/.

[8] Ros Rodrigo, José Manuel: *Homología persistente como herramienta de análisis de redes neuronales*, 2022. https://github.com/joros244/TFGMath2022.

[9] Sedgewick, Robert and Kevin Wayne: *Algorithms, 4th Edition.* Addison-Wesley, 2011, ISBN 978-0-321-57351-3.

[10] The GUDHI Project: *GUDHI User and Reference Manual*. GUDHI Editorial Board, 2015. http://gudhi.gforge.inria.fr/doc/latest/.

[11] Van Rossum, Guido and Fred L. Drake: *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009, ISBN 1441412697.

[12] Watanabe, Satoru and Hayato Yamana: *Topological measurement of deep neural networks using persistent homology*, Jan 2022. ISSN 1573-7470. https://doi.org/10.1007/s10472-021-09761-3.