Trabajo de Fin de Grado

Framework para la gestión del ciclo de vida de las redes neuronales

TITULACIÓN: Grado en Ciencia e Ingeniería de Datos

AUTOR: Joel Del Rosario Pérez

TUTORIZADO POR
Octavio Roncal Andrés

Junio 2024

Agradecimientos

A mis padres

Por acompañarme y sostenerme en cada paso de mi vida

Por enseñarme cada día a seguir intentándolo sin importar las barreras

Por creer en mi en todo momento, sin importar la situación

En resumen, yo más.

Resumen

Dentro del ámbito de la Inteligencia Artificial, se pueden encontrar muchos problemas que dificultan el uso de los frameworks actuales dentro del Deep Learning, como la falta de automatización de muchos procesos repetitivos o la necesidad de muchos conocimientos previos para generar modelos sencillos. Con el objetivo de solventar estos obstáculos, se ha desarrollado un framework para la gestión del ciclo de vida de una red neuronal que optimiza y mecaniza muchos procedimientos típicos de esta rama. Este framework forma parte de un sistema mayor, llamado Flogo, que hará de las redes neuronales una tecnología mucho más accesible para los programadores.

Abstract

Within the field of Artificial Intelligence, many problems can be found that hinder the use of current frameworks within Deep Learning, such as the lack of automation of many repetitive processes or the need for a lot of prior knowledge to generate simple models. With the aim of solving these obstacles, a framework has been developed for managing the life cycle of a neural network that optimizes and mechanizes many typical procedures of this branch. This framework is part of a larger system, called Flogo, that will make neural networks a much more accessible technology for programmers.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

> La inteligencia en las máquinas será la última invención que el ser humano necesite hacer
>
> Nick Bostrom

En el panorama actual, marcado por la rápida evolución tecnológica, la Inteligencia Artificial (IA) se ha posicionado como una disciplina fundamental, revolucionando la forma de trabajar en diversos sectores. Su impacto transversal y su potencial para transformar industrias la convierten en un tema de gran relevancia, especialmente para quienes se preparan para ingresar al mercado laboral.

En el ámbito informático, y particularmente en el campo de la IA, es común encontrar términos en inglés debido a su amplia adopción y la falta de traducciones adecuadas para muchos de ellos. En esta memoria, se utilizarán algunos de estos términos anglicismos y acrónimos de anglicismos, los cuales se explicarán en detalle en el glosario al final del documento para facilitar la comprensión de aquellos lectores que lo necesiten. La utilización de anglicismos se justifica por las siguientes razones:

- **Precisión:** En algunos casos, no existe una traducción al español que capture con exactitud el significado del término en inglés.
- **Universalidad:** Muchos términos anglicismos son ampliamente utilizados en la comunidad internacional de IA, lo que facilita la comunicación y la comprensión entre investigadores y profesionales.
- **Eficiencia:** El uso de términos en inglés evita la necesidad de crear traducciones imprecisas o poco utilizadas.

La inteligencia artificial (IA) se destaca como un campo revolucionario en la informática, impulsado por avances tecnológicos que permiten la ejecución de algoritmos complejos. Su impacto abarca diversos sectores, desde la salud hasta el entretenimiento, optimizando procesos y fomentando la innovación.

Desde una perspectiva científica, la IA atrae por sus rápidos avances, tanto teóricos como prácticos. La inversión en investigación y desarrollo es considerable, impulsada por el enorme potencial que aún posee esta tecnología. Además, es uno de los temas fundamentales tratados en el grado, lo que permite adquirir un conocimiento extenso sobre el tema.

Dentro de los diversos ámbitos que cubre esta disciplina, el Deep Learning (DL) se destaca como uno de sus componentes más prometedores, debido principalmente a su capacidad para obtener resultados superiores en procesos de aprendizaje y predicción a partir de grandes conjuntos de datos, superando significativamente a los métodos tradicionales. Una de las técnicas fundamentales dentro del DL son las redes neuronales.

Las redes neuronales son modelos computacionales inspirados en la estructura y funcionamiento del cerebro humano, diseñados para procesar información mediante las conexiones entre las neuronas de cada capa. Aunque fueron concebidas teóricamente muchos años antes, no se implementaron ni probaron hasta que la tecnología avanzó lo suficiente para permitir su uso eficiente.

Durante el transcurso del grado, los estudios enfocados en la IA han permitido identificar varios desafíos significativos en el desarrollo y mantenimiento de redes neuronales. Estos desafíos incluyen la falta de automatización y estandarización en numerosos procesos, la fuerte dependencia a los *frameworks* disponibles en el mercado como Pytorch[22] y *TensorFlow*[2], y la necesidad de poseer una base de conocimientos considerable incluso para realizar tareas básicas.

Para abordar las dificultades identificadas en el desarrollo y mantenimiento de redes neuronales, se ha desarrollado *Flogo*. Este sistema está diseñado para facilitar la creación de arquitecturas de redes neuronales y la gestión de su ciclo de vida. *Flogo* incluye un lenguaje específico de dominio (DSL), un *framework* para el diseño de redes neuronales y otro *framework* [25] para la administración de sus procesos principales, como el entrenamiento, almacenamiento y testeo. Cada una de estas partes ha sido desarrollada en un Trabajo Final de Título (TFT) distinto y se estructura tal y como se indica en la Ilustración 1.1.

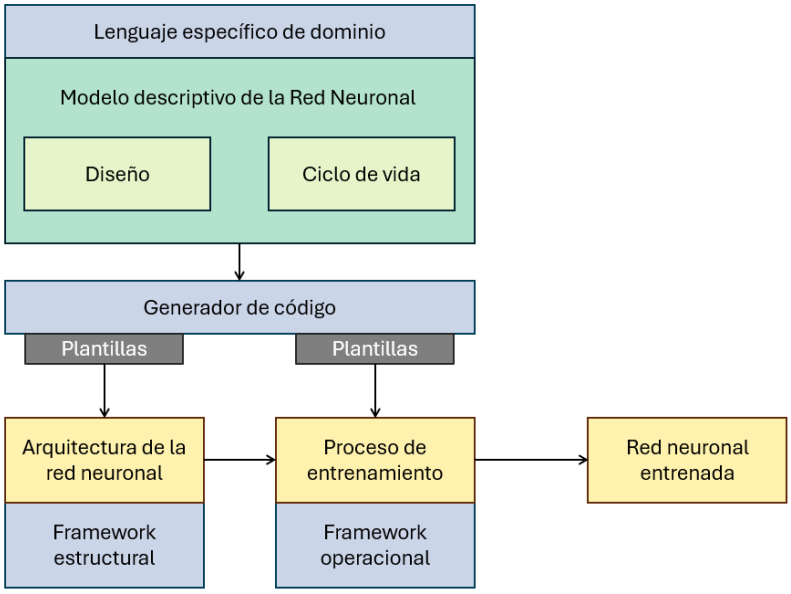

Ilustración 1.1: Diagrama del sistema completo de *Flogo*

En ella, podemos observar cómo el sistema se inicia a partir del DSL, que es usado por el usuario para crear un modelo descriptivo de la arquitectura de la red neuronal y el ciclo de vida que se quiere que esta siga. El generador de código de *Flogo* será quien, a partir del modelo especificado por el usuario y a través de las plantillas creadas con *ItRules*, creará los archivos Python [11] necesarios para ejecutar, usando los dos *frameworks* elaborados, los pasos indicados por el desarrollador.

El código generado por el *framework* estructural se utilizará para definir la estructura en capas que debe tener el modelo resultante, mientras que el código del *framework* operacional definirá el proceso de entrenamiento de la arquitectura creada. A partir de la ejecución de todo este código, se obtendrá una red neuronal con pesos aprendidos y lista para ser usada.

Este TFT ha estado centrado en la creación del *framework* operacional y el servicio de laboratorios [26]. La función del *framework* es automatizar las etapas más repetitivas en el ciclo de vida de las redes neuronales. Con él, se busca aumentar la eficiencia en los procesos de entrenamiento de modelos adaptados a los problemas específicos que pueda tener el usuario, y permitir que estos procesos puedan ser usado por personas con menos conocimientos sobre DL.

El servicio de laboratorios tiene como objetivo facilitar la gestión de todas las pruebas de entrenamiento de modelos que el usuario desee realizar con los *frameworks* que conforman *Flogo*. Para ello, permite a los desarrolladores disponer de su propio repositorio de arquitecturas, laboratorios y *datasets*, y ejecutar tantas pruebas como consideren necesarias. Asimismo, este servicio se encargará de administrar todos los resultados generados durante el entrenamiento de una arquitectura, incluyendo los modelos y sus estadísticas.

Antes de abordar en profundidad el trabajo realizado, se profundizará en la problemática encontrada. Se explicará en detalle qué es *Flogo*, cuáles fueron las motivaciones detrás de su

desarrollo y cómo se divide, proporcionando una visión general de las partes no abordadas en este trabajo.

Asimismo, es fundamental establecer el contexto y los objetivos iniciales del proyecto. Por ello, se analizarán otros productos o investigaciones en la comunidad científica que podrían satisfacer las necesidades identificadas. Posteriormente, se explorarán las metas del proyecto, identificando las competencias adquiridas durante el programa académico que fueron esenciales para su desarrollo.

Seguidamente, se presentará el producto resultante de este TFT: el *framework* para la gestión del ciclo de vida de una red neuronal y el servicio de laboratorio, destinado a la administración de experimentos. Se explicarán sus conceptos principales y se ofrecerá orientación sobre su uso por parte del usuario final. Además, se detallarán las herramientas y metodologías utilizadas en el proyecto, junto con una cronología de los pasos seguidos para alcanzar la solución final.

Finalmente, se presentarán las conclusiones del trabajo, destacando las contribuciones de *Flogo* a la comunidad científica, el grado de cumplimiento de los objetivos iniciales, propuestas para trabajos futuros que podrían mejorar el resultado final y los principales aprendizajes obtenidos durante la realización de este proyecto.

Capítulo 2

Presentación del problema

Durante los años en la universidad, se ha adquirido una base teórica y práctica en el ámbito de la inteligencia artificial (IA) y, más específicamente, en el aprendizaje profundo (DL). Estos conocimientos han permitido identificar diversas necesidades y problemas comunes en estos campos, así como desarrollar la capacidad para proponer soluciones a estos desafíos.

La IA es una rama de investigación moderna que ha experimentado grandes avances en los últimos años y que aún posee un enorme potencial por explorar. Actualmente, el sector prioriza la producción de modelos útiles en el mercado laboral por encima de la creación de código sostenible y escalable, adaptado a futuros cambios. Esto se debe a que es una disciplina en constante evolución y transformación. Como resultado, a menudo no se aplican las técnicas y patrones más importantes de ingeniería de software en el proceso de creación de redes neuronales funcionales, lo que genera complicaciones en términos de eficiencia y generalización de conceptos.

Entre las diversas dificultades identificadas, una de las más destacadas es la falta de automatización en muchos de los procesos clave dentro de la creación y entrenamiento de redes neuronales. Actualmente, existen numerosas tareas en DL que resultan repetitivas o tediosas para los desarrolladores, como probar una misma arquitectura con diferentes hiperparámetros para determinar cuáles ofrecen mejores resultados, o definir capa por capa una red neuronal. Además, si no se sigue una buena metodología, este procedimiento puede dar lugar a errores significativos provocados por simples descuidos, como descartar el mejor conjunto de hiperparámetros debido a una mala administración de los resultados obtenidos.

Otro desafío importante es la dependencia de muchas empresas, proyectos y programadores a las herramientas específicas en las que se formaron, como Pytorch o TensorFlow. Estas herramientas tienden a ser altamente complejas y a usar una sintaxis propia, a menudo descuidando conceptos básicos y generales que podrían proporcionar a los usuarios un conocimiento más completo del ámbito en el que trabajan. Esto puede resultar en que las personas se encuentren limitadas a la herramienta que aprendieron inicialmente, ya que tanto el proceso de aprendizaje de una nueva como la migración de todas sus redes neuronales a otro *framework* son tareas complicadas.

En la figura Ilustración 2.1 se puede ver un claro ejemplo de un proceso de entrenamiento de una red neuronal muy sencillo creado con Pytorch. En él, se observan estos aspectos comentados sobre el uso de una sintaxis compleja y poco legible, que dificulta, especialmente para personas con menos conocimientos, la comprensión del código. En la mayor parte de las instrucciones, se ejecutan procedimientos muy específicos del *framework* utilizado, que no deberían ser responsabilidad de los usuarios menos expertos en la materia.

```
dataset = np.loadtxt(DATASET_PATH, delimiter=',')
X = dataset[:,0:8]
y = dataset[:,8]
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

loss_fn = torch.nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
n_epochs = 100
batch_size = 10

for epoch in range(n_epochs):
    for i in range(0, len(X), batch_size):
        x_batch = X[i:i + batch_size]
        y_prediction = model(x_batch)
        y_batch = y[i:i + batch_size]
        loss = loss_fn(y_prediction, y_batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print(f'Finished epoch {epoch}, latest loss {loss}')
```

Ilustración 2.1: Código de ejemplo usado para entrenar una red neuronal con Pytorch

La complejidad de los entornos de desarrollo de redes neuronales limita su accesibilidad para la mayoría de los programadores, ya que se requieren amplios conocimientos sobre inteligencia artificial para lograr un producto mínimo viable. Aunque un profundo entendimiento del tema es crucial para entrenar modelos más complejos o alcanzar resultados óptimos, no debería ser indispensable para aquellos que buscan soluciones a problemas más simples.

Otro aspecto a destacar en estos *frameworks* de DL es la limitada capacidad que ofrecen a los usuarios para desarrollar programas que se adhieran a los principios fundamentales de código limpio, eficiencia y comprensión para terceros. Como se evidencia en la Ilustración 2.2, que representa el código necesario para definir una red neuronal básica con Pytorch, se omiten muchas recomendaciones básicas para obtener un programa legible y sostenible a lo largo del tiempo. Entre estas recomendaciones se incluyen evitar la repetición de código y utilizar nombres significativos para las variables.

```python
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5)
        self.fc1 = nn.Linear(in_features=16 * 5 * 5, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        self.fc3 = nn.Linear(in_features=84, out_features=10)

    def forward(self, network_input):
        c1 = F.relu(self.conv1(network_input))
        s2 = F.max_pool2d(c1, kernel_size=(2, 2))
        c3 = F.relu(self.conv2(s2))
        s4 = F.max_pool2d(c3, kernel_size=2)
        s4 = torch.flatten(s4, 1)
        f5 = F.relu(self.fc1(s4))
        f6 = F.relu(self.fc2(f5))

        return self.fc3(f6)
```

Ilustración 2.2: Código de ejemplo usado para definir la arquitectura de una red neuronal con Pytorch

Este fenómeno resulta en arquitecturas de modelos que tienden a ser poco comprensibles, incluso para personas con amplia experiencia en el *framework*. Esto dificulta considerablemente la difusión de las redes y modelos obtenidos, lo que ralentiza el desarrollo del sector en comparación con el uso de un estándar común para toda la comunidad científica.

Además, la gestión de las pruebas realizadas por muchos programadores de IA suele ser bastante errática. Actualmente, no existe un estándar o metodología clara sobre cómo llevar a cabo todas las pruebas necesarias con conjuntos de hiperparámetros distintos para obtener el mejor rendimiento posible de una arquitectura determinada. La falta de estructuración en este procedimiento tiende a aumentar los tiempos y recursos necesarios para la obtención de modelos funcionales.

En definitiva, se han identificado múltiples problemas en el proceso convencional de desarrollo de modelos de DL, lo que hace que estas actividades sean más repetitivas, más propensas a errores y más difíciles de entender para desarrolladores con menos experiencia en el tema. En respuesta a estas dificultades, se han desarrollado una serie de soluciones con el objetivo de eliminarlas o reducirlas a niveles aceptables.

Capítulo 3

Contexto

En este capítulo se abordarán cuatro temas principales. Primero, se examinarán los trabajos existentes en la comunidad científica y en la sociedad que guardan relación con nuestro proyecto, destacando su carácter novedoso e innovador. Luego, se expondrán los objetivos y motivaciones iniciales al comenzar este Trabajo Final de Título (TFT). Posteriormente, se analizarán las competencias específicas adquiridas durante el grado que fueron esenciales para la elaboración de este proyecto y, para concluir, se describirá en que consiste el sistema de *Flogo*.

3.1. Estado actual

Esta sección tiene como objetivo proporcionar una visión general de la situación en el campo de la IA en relación con las herramientas que facilitan a los usuarios la creación y el entrenamiento de redes neuronales. Se explorará el estado del arte en busca de software o artículos que propongan soluciones similares a las que se detallarán más adelante, o bien para determinar si estamos ante un producto verdaderamente innovador. Para ello, se llevó a cabo una investigación exhaustiva sobre las soluciones actuales aplicables a problemas similares.

En esta investigación se exploró GitHub Actions [1], un servicio de automatización incorporado en la plataforma GitHub. Este servicio permite a los desarrolladores automatizar tareas relacionadas con el desarrollo, pruebas y despliegue de software directamente desde sus repositorios en GitHub. Mediante GitHub Actions, es posible crear flujos de trabajo personalizados, conocidos como *workflows*. Estos se configuran a través de archivos en formato *Ain't Markup Language* (YAML) [10], los cuales definen las acciones a ejecutar en respuesta a eventos específicos o condiciones dentro del repositorio.

La necesidad de automatizar ciertos procesos llevó a GitHub a desarrollar esta herramienta, lo que inspiró la creación de metodologías para abordar aspectos críticos de problemas similares en diversos campos.

En el ámbito del DL, un ejemplo similar es Keras [7], una biblioteca de código abierto

diseñada para el desarrollo de modelos de aprendizaje profundo. Keras se caracteriza por su modularidad, extensibilidad y facilidad de uso. Funcionando sobre plataformas de bajo nivel como TensorFlow o Theano, proporciona una interfaz de alto nivel que facilita la configuración, entrenamiento y evaluación de redes neuronales profundas.

En su definición, se incluyen varios aspectos fundamentales en la solución al problema abordado en esta memoria, como el desarrollo de redes neuronales con una sintaxis menos compleja, comparado con otros entornos disponibles en el mercado. De esta forma, Keras logra reducir la complejidad de muchas de las actividades básicas en el entrenamiento de modelos y disminuir los conocimientos necesarios para afrontar un problema de este tipo. Sin embargo, este *framework* no posee otras características que permitan dar solución de forma completa al problema presentado, ya que no se logra automatizar la mayor parte de los procesos ni reducir la cantidad de código repetido que puede llegar a provocar errores. Por tanto, a pesar de ser una buena aproximación a lo que se busca, no cumple con todos los requisitos.

Este proceso de abstracción y generalización que se quiere llevar a cabo no es nuevo. En 2023, *Hira Naveed, Chetan Arora, Hourieh Khalajzadeh, John Grundy* y *Omar Haggag*, siguiendo motivaciones similares a las de esta memoria, elaboraron un estudio donde, a partir de la lectura y análisis de 3934 *papers* distintos, destacaron las principales tendencias y futuras líneas de trabajo relacionadas con el uso del enfoque de *Model Driven Engineering* dentro del ámbito de DL. En él, se recalcó la escasa cantidad de estudios relaciones con dicho tema, entre otras muchas conclusiones, a pesar de los beneficios y posibles ventajas que este enfoque podría aportar [21].

Otro componente software relacionado con la solución elaborada que debemos mencionar es *Data Version Control* (DVC) [28] . Esta es una herramienta de gestión de versiones y flujo de trabajo diseñada para proyectos de ciencia de datos y aprendizaje automático. Similar a *Git*, permite a sus usuarios almacenar y versionar datos, modelos y configuraciones de experimentos. Esto se logra mediante la creación de archivos de configuración y metadatos que registran los cambios y permiten la reproducción de experimentos, facilitando la colaboración y la reproducibilidad en equipos de ciencia de datos.

Como se ha visto, existen varios productos de software y estudios que intentan cubrir necesidades similares. La problemática tratada ha sido identificada por muchos investigadores y ha cobrado gran importancia en los últimos años gracias al ascenso de la IA como herramienta. Sin embargo, no se ha encontrado ningún artículo científico o software desarrollado con los mismos objetivos que se plantean en esta memoria. Por lo tanto, se puede afirmar que se ha creado un *framework* original que aporta valor a la sociedad y al mercado actual.

3.2. Objetivos iniciales

La idea de este proyecto surgió a partir de la identificación de ciertas necesidades en el trabajo diario con redes neuronales. Inicialmente, se propuso elaborar un sistema que hiciera del proceso de creación y gestión de modelos de DL, un procedimiento más sencillo, agradable

y rápido para los desarrolladores. Para ello, se plantearon una serie de objetivos iniciales que debían cumplirse.

El objetivo más general e importante del proyecto era crear un *framework* avanzado y altamente eficiente para el entrenamiento de redes neuronales, ofreciendo funcionalidades robustas. Este *framework* debía ser accesible para usuarios de diferentes niveles de conocimiento, no solo para expertos en la materia. Además, debía permitir la optimización y personalización eficaz de los procesos de entrenamiento de redes neuronales en diversos contextos y aplicaciones. Junto a este objetivo principal, se establecieron otros objetivos específicos que se intentarían cumplir:

- **Diseño modular y extensible:** Resultaba fundamental lograr un diseño modular y extensible, que permitiera a los usuarios agregar o modificar componentes según fuera necesario. De esta forma, se aseguraba que el *framework* fuera escalable y pudiera adaptarse a futuras tecnologías y metodologías en el campo de las redes neuronales.
- **Interfaz sencilla e intuitiva:** Se propuso elaborar una interfaz de aplicaciones que fuera intuitiva y fácil de usar, reduciendo la curva de aprendizaje para los nuevos usuarios. Para ello, era esencial proporcionar documentación detallada y ejemplos de uso, facilitando la adopción del *framework* por parte de nuevos desarrolladores.
- **Automatización y optimización de hiper-parámetros:** Se planteó que el producto final debía alcanzar un cierto grado de automatización y optimización en la asignación de los hiper-parámetros necesarios para el entrenamiento de una red neuronal. Para lograrlo, se implementarían herramientas para su ajuste automático, mejorando la eficiencia y efectividad del proceso. Además, se buscaba facilitar la experimentación y la comparación de diferentes configuraciones de hiper-parámetros.
- ***Framework* actualizado:** Se estableció como importante lograr un *framework* que implementara las últimas técnicas y algoritmos para optimizar el rendimiento de las redes neuronales. Además, debía permitir la personalización de estrategias de entrenamiento para adaptarse a diferentes tipos de redes neuronales y conjuntos de datos.
- **Compatibilidad entre redes:** Actualmente, existen una gran cantidad de tipos distintos de redes neuronales. Por ello, se enfatizó en desarrollar un sistema que asegurara la compatibilidad con una amplia gama de arquitecturas, incluyendo redes convolucionales, recurrentes y generativas. Además, era esencial proveer funcionalidades para facilitar el entrenamiento de modelos complejos y de gran escala.
- **Evaluación y análisis de modelos:** La evaluación y el análisis de los modelos generados son partes esenciales del procedimiento de entrenamiento. Por ello, se estableció como objetivo incorporar herramientas para el análisis detallado y la evaluación del rendimiento de los modelos entrenados, proporcionando métricas y visualizaciones que ayuden a los usuarios a entender y mejorar sus modelos y resultados.
- **Compatibilidad e integración con herramientas existentes:** Era importante diseñar el *framework* para que fuese compatible con bibliotecas y herramientas estándar en el ámbito de la ciencia de datos y el DL. De esta forma, se facilitaría la integración

con otros sistemas y plataformas para extender su funcionalidad y aplicabilidad cuando se requiriera o surgiera una nueva tecnología en este ámbito.

En el Cuadro 3.1, se podrán ver en forma de tabla los objetivos explicados anteriormente:

Cuadro 3.1: Objetivos iniciales del proyecto

Referencia	**Objetivo**
Objetivo 1	Diseño modular y extensible
Objetivo 2	Interfaz sencilla e intuitiva
Objetivo 3	Automatización y optimización de hiper-parámetros
Objetivo 4	Framework actualizado
Objetivo 5	Compatibilidad entre redes
Objetivo 6	Evaluación y análisis de modelos:
Objetivo 7	Compatibilidad e integración con herramientas existentes:

En resumen, el cumplimiento de estos objetivos específicos conduciría al desarrollo de un *framework* de entrenamiento de redes neuronales que no solo sea poderoso y flexible, sino también accesible y amigable para el usuario, promoviendo la eficiencia y la innovación en el campo del DL. Partiendo de estas ideas iniciales, comenzamos a desarrollar nuestro proyecto, culminando en el producto que se detallará en los siguientes capítulos.

3.3. Competencias específicas

Para la elaboración de esta tarea, han sido necesarios muchos de los conocimientos y enseñanzas adquiridas durante estos 4 años, puesto que es un proyecto que engloba muchos campos trabajados durante las diferentes asignaturas del grado, como IA o desarrollo de software. Por esa razón, muchas de las competencias específicas relacionadas con nuestro grado han sido determinantes para alcanzar este resultado final. Sin embargo, destacamos dos sobre el resto:

- **ED1:** La competencia específica *ED1* es enunciada de la siguiente forma. *Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación.* Esta ha sido cubierta en este *TFT* mediante la implementación de diversas técnicas propias de la Ciencia de Datos dentro del *framework* final, como las funciones de pérdidas y los optimizadores. Posteriormente, se explicará cómo se han implementado y qué función cumplen dentro del entorno.
- **ED2:** Esta competencia dice lo siguiente. *Capacidad para adquirir, obtener, formalizar y representar el conocimiento humano en una forma computable para la resolución de*

problemas mediante un sistema informático en cualquier ámbito de aplicación, particularmente los relacionados con aspectos de computación, percepción y actuación en ambientes o entornos inteligentes. Esta competencia está estrechamente vinculada al proyecto final. Para desarrollar un marco de trabajo completo y funcional, fue necesario adquirir conocimientos previos sobre las soluciones disponibles actualmente. Esto permitió identificar las ideas recurrentes en todos los métodos y las debilidades comunes que se presentaban.

- **ED4:** Esta se define de la siguiente manera. *Capacidad de identificar y analizar problemas y diseñar, desarrollar, implementar, verificar y documentar soluciones software en ámbitos de aplicación de la Inteligencia Artificial en Ciencia e Ingeniería de Datos.* Su relación con el proyecto es evidente, ya que se ha identificado una problemática común en el trabajo diario de los científicos de datos. Tras un análisis exhaustivo de esta situación, se ha implementado una solución de software para mitigar, o incluso eliminar, el impacto negativo detectado, cumpliendo así con los requisitos de la competencia.
- **ED6:** A continuación, se expone la competencia. *Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos utilizados en Ciencia de Datos, particularmente las relacionadas con el análisis, predicción y prospectiva de grandes volúmenes de datos.* En el producto final, se pueden ver reflejadas muchas de los algoritmos fundamentales y actuales del DL, una de las ramas principales dentro de la Ciencia de Datos. Por ello, podemos asegurar que esta competencia específica ha sido determinante, puesto que sin el conocimiento de estas técnicas, el *framework* hubiese quedado incompleto.
- **EF3:** La competencia *EF3* es descrita como *conocimientos básicos sobre el uso y programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación de la ingeniería.* Esta ha sido fundamental para el desarrollo de este trabajo, puesto que los conocimientos básico sobre programación y ordenadores han sido fundamentales para poder alcanzar una solución óptima y viable a la problemática presentada.
- **EC4:** Esta competencia, común a toda la rama de informática, dice lo siguiente. *Conocimiento y aplicación de los principios, metodologías y ciclos de vida de la ingeniería del software.* En este proyecto, se busca tener una pequeña unión de dos de los campos más importantes que han sido vistos en el grado: la IA y la Ingeniería del Software. Por ello, esta competencia resulta fundamental y su aplicación queda ampliamente demostrada en este proyecto.

3.4. Flogo

Flogo aborda la problemática presentada mediante la creación de diferentes componentes, cada uno diseñado para solventar o reducir las dificultades identificadas. Uno de los aspectos más destacados del problema es la necesidad de simplificar el uso de herramientas de DL a los usuarios, permitiéndoles modificar solo los aspectos esenciales. Para esto, se desarrolló un

Lenguaje Específico de Dominio (DSL).

Un DSL es un lenguaje de programación especializado diseñado para un conjunto particular de tareas o una industria específica. Esto lo diferencia de los lenguajes de propósito general, como Python [11] o Java [8], que están destinados a una amplia variedad de aplicaciones. Los DSL se crean para simplificar y optimizar el desarrollo dentro de su dominio específico, proporcionando abstracciones y sintaxis que se alinean directamente con los conceptos y necesidades del campo en cuestión. Esto puede hacer que las tareas complejas sean más comprensibles para los usuarios que trabajan en ese dominio, al ser más intuitivo que un lenguaje de programación tradicional y reflejar directamente las problemáticas y soluciones comunes.

El uso de un DSL en *Flogo* permite abordar algunos de los principales desafíos planteados. Por un lado, se crea una sintaxis mucho más sencilla, lo que facilita a los investigadores comparar y compartir arquitecturas, ya que estas son más legibles en comparación con otros *frameworks* tradicionales. Por otro lado, se obtiene una interfaz de uso que requiere mucho menos conocimiento previo para ser utilizada. Con un pequeño entendimiento de los conceptos más básicos de DL, los usuarios pueden desarrollar modelos listos para ser usados, haciendo el proceso más accesible y eficiente.

```
Laboratory(epochs = 10, name = "WineQuality")
    SGD(lr=0.0001, momentum=0, momentumDecay=0, weightDecay=0)
    MSELoss
    RegressionStrategy
    LossDrivenEarlyStopper(10, 0.01)
    Dataset(name = "winequality-red", batchSize=10)
        Split(train = 0.7, test = 0.2, validation = 0.1)

    Experiment b525
        Materialization(vLayer="01") > BatchNormalization(eps=0.00001, momentum=0.3)
        Materialization(vLayer="02") > LogSigmoid
        Materialization(vLayer="03") > Dropout(probability=0.6)

Architecture WineQualityNeuralNetwork
    LinearSection
        Input(x=11)
        Block
            Linear > Output(x=10)
            VLayer(id="01")
            VLayer(id="02")
            VLayer(id="03")
        Block
            Linear > Output(x=1)
            VLayer("02")
```

Ilustración 3.1: Ejemplo de uso del DSL

En la Ilustración 3.1, se muestra un ejemplo de uso del DSL, que presenta muchos de los conceptos principales, como *Architecture*, *Laboratory*, y *Experiment*. El primero de los

mencionados se usa para definir las redes neuronales, mientras que *Laboratory* y *Experiment* se utilizan para definir su entrenamiento. Como se puede observar, utiliza un léxico sencillo y legible, incluso para personas con menos conocimientos sobre el tema. A lo largo del documento, se explicarán la mayoría de estos objetos.

En lo que respecta al *framework* estructural y al *framework* operacional, sus características ofrecen numerosos beneficios al sistema en su totalidad. Estos *frameworks* fueron diseñados para adaptarse a posibles cambios en la tecnología utilizada para la implementación inicial, logrando así una completa abstracción. De esta manera, se evita cualquier dependencia con una tecnología específica, resultando en *frameworks* escalables y resistentes a cambios.

Para lograr esto, se llevó a cabo un proceso de generalización de todos los conceptos esenciales en el diseño y entrenamiento de redes neuronales. Los *frameworks* utilizan estos conceptos esenciales para minimizar la cantidad de parámetros y variaciones no esenciales presentadas a los usuarios. De esta forma, los desarrolladores pueden evitar la necesidad de aprender muchos conceptos específicos del *framework* que estén utilizando, y centrarse en aquellos que realmente les ayudan a comprender mejor el funcionamiento de sus modelos.

Otro de los principales objetivos perseguidos con estas bibliotecas es la optimización de los procesos de creación de modelos. A menudo, se otorga un grado de libertad al investigador para definir sus propias configuraciones en el entrenamiento o en el diseño del modelo. Estos *frameworks* reducen dichas posibilidades para alcanzar la máxima eficiencia en los procesos y disminuir las responsabilidades del usuario, facilitando así la creación de modelos efectivos y eficientes.

Por último, es importante destacar las principales motivaciones que impulsaron el desarrollo del servicio de laboratorio. Como se mencionó anteriormente, una de las dificultades más significativas es la falta de organización en los procesos de prueba de diversas arquitecturas y conjuntos de hiperparámetros. La creación de este servicio busca asistir al usuario en la gestión eficiente de todas las pruebas que desee realizar, facilitando aspectos como el almacenamiento de resultados, la organización de pruebas, y la conservación de modelos entrenados, entre otros.

En resumen, *Flogo* logra su objetivo de resolver, o al menos reducir significativamente, la mayoría de las dificultades identificadas en el problema inicial. Cada componente de *Flogo* desempeña una función esencial, contribuyendo al correcto funcionamiento del sistema en su conjunto. *Flogo* simplifica el aprendizaje y la experimentación con algoritmos de DL al minimizar conceptos innecesarios para el usuario ofreciendo una interfaz sencilla. Además, facilita la comunicación de arquitecturas y resultados al abstraerse de los conceptos específicos de los *frameworks* actuales en el mercado, proporcionando una herramienta más accesible y eficaz para los desarrolladores.

Capítulo 4

Framework operacional

Como se vio en el capítulo de Presentación del problema, en este documento se quiere encontrar soluciones a varias de las dificultades que se repiten con frecuencia en el proceso de creación de modelos válidos y funcionales, como la repetición de muchos procedimientos básicos de forma continua y la complejidad de muchas de las tecnologías disponibles actualmente en el mercado. A partir de un análisis de la problemática, se llegó a la conclusión de que la elaboración de un sistema completo, compuesto por dos *frameworks* y un DSL, sería la mejor solución para todos los desafíos encontrados.

Uno de los puntos fundamentales de *Flogo* ha sido el desarrollo del *framework* para la gestión del ciclo de vida, cuyo objetivo principal es simplificar y automatizar todos los procesos que engloban la administración de una red neuronal, como su entrenamiento, almacenamiento, testeo, etc. Este permitirá a sus usuarios abstraerse de los *frameworks* actuales y tener un entorno de desarrollo sencillo y ajeno a conceptos propios de una tecnología específica de DL.

Para desarrollar este *framework*, fue necesario abstraer los conceptos más generales y comunes en los procedimientos de entrenamiento tradicionales. Para lograrlo, se analizaron las diversas formas en que la comunidad científica lleva a cabo este proceso, observando muchas metodologías y tecnologías de DL distintas. Es importante destacar que fue crucial explorar una amplia variedad de técnicas para identificar qué métodos o actuaciones son comunes y se repiten de forma sistemática en todos los programas, evitando quedar limitados por una única perspectiva. De esta manera, se logró alcanzar una conceptualización de todo el proceso de gestión de las redes neuronales, lo que permitió simplificarlo y automatizarlo.

4.1. Conceptos del dominio

A través del proceso de abstracción, se identificaron una serie de conceptos recurrentes en la mayoría de los procedimientos examinados, los cuales constituyen una parte fundamental de los mismos. Estos conceptos son:

4.1.1. Modelo

Previa a la definición de modelo, es importante recordar el significado de arquitectura dentro del contexto de *Flogo*. La arquitectura se refiere a la definición estructural de la red neuronal. En este sentido, un modelo se refiere a una arquitectura en la que las capas ya tienen asignados pesos, ya sea de manera aleatoria o como resultado de un proceso de entrenamiento. Por lo tanto, la distinción principal entre arquitectura y modelo radica en la presencia o ausencia de pesos en las capas.

4.1.2. Función de pérdidas

Una función de pérdida, también conocida como función objetivo o función de coste, es una medida cuantitativa que evalúa el rendimiento de un modelo de DL en una tarea específica. Esta función calcula las diferencias existentes entre las predicciones del modelo y los valores reales o esperados de los datos de entrenamiento. Esto permite que el algoritmo de optimización ajuste los parámetros o pesos del modelo para reducir dichas diferencias y mejorar su capacidad para generalizar a datos no vistos. Es un concepto fundamental para el aprendizaje de redes neuronales, ya que permite conocer el rendimiento de un modelo a través de un valor numérico.

Actualmente, existen una gran cantidad de funciones de pérdida, principalmente debido a la diversidad de problemas que enfrenta el aprendizaje automático. Cada una de ellas tiene atributos únicos que las hacen más efectivas en diferentes contextos, por lo que es importante seleccionar la más adecuada para cada situación. El *framework* da soporte a una amplia variedad de funciones de pérdida. Las más destacadas están explicadas en Funciones de pérdidas.

4.1.3. Optimizador

Una vez obtenida una medida que indique el rendimiento de la red neuronal al intentar predecir las salidas a partir de unos valores de entrada, se debe lograr que esta aprenda de sus errores y obtenga mejores resultados en futuras iteraciones. Estos resultados están determinados por los valores de cada uno de los parámetros que componen el modelo. Por lo tanto, para mejorar la precisión obtenida, es necesario realizar modificaciones en estos parámetros. Con este objetivo, surge el concepto del optimizador.

Un optimizador es un algoritmo utilizado durante el entrenamiento de una red neuronal para ajustar los pesos y sesgos de sus conexiones, con el objetivo de minimizar una función de pérdida. En otras palabras, es la función encargada de determinar los nuevos valores de los parámetros para lograr un mejor rendimiento del modelo. Existen numerosas alternativas o variantes de optimizadores, aunque en el fondo todos comparten una base común. [3]

Una función de pérdida indica el rendimiento general del modelo. Cuanto menor sea su valor resultante, mejores serán los valores obtenidos. Por esta razón, el objetivo es minimizar

al máximo dicha función a través de algoritmos de optimización. Uno de los algoritmos más comunes para llevar a cabo esta tarea es el descenso por gradiente.

Existe una gran variedad de optimizadores en el mundo del DL, y las herramientas disponibles en el mercado actual dan soporte a la mayoría de ellos. Cada uno tiene sus propias ventajas e inconvenientes, aportando al desarrollador funcionalidades y características distintas. Los optimizadores incluidos en nuestro *framework* se pueden encontrar en Optimizadores.

4.1.4. Device

El proceso de entrenamiento de una red neuronal es altamente demandante en términos de recursos computacionales, lo que lo convierte en una tarea costosa. Este fue uno de los principales impedimentos que limitaron la aplicación práctica de las redes neuronales en el pasado. Sin embargo, con el avance tecnológico y el desarrollo de hardware más potente, los investigadores han podido poner a prueba sus hipótesis de manera más efectiva.

Uno de los avances significativos en el ámbito del hardware para ejecutar redes neuronales fue el desarrollo de las primeras unidades de procesamiento gráfico (GPU). Estas unidades se caracterizan por su capacidad para realizar operaciones matemáticas a alta velocidad. Dado que los procesos de entrenamiento de modelos de redes neuronales implican un gran número de operaciones, el uso de GPUs puede reducir significativamente los tiempos de ejecución. Sin embargo, las GPUs suelen ser costosas y no todos los programadores tienen acceso a ellas.

El concepto de *Device* fue desarrollado con el objetivo de brindar a los usuarios la posibilidad de elegir en qué procesador se ejecutará el entrenamiento seleccionado. Es importante que los desarrolladores tengan esta opción, ya que los tiempos y los recursos consumidos pueden variar considerablemente según el dispositivo de ejecución. Actualmente, los dispositivos disponibles para mandar la ejecución de código son la GPU, la unidad central de procesamiento (CPU) el *Map Production System* (MPS) del ordenador.

4.1.5. Stopper

Una técnica muy habitual en el ámbito del DL es detener el entrenamiento de los modelos cuando se tiene certeza de que ya no están mejorando más. Esto se lleva a cabo por dos razones principales. El primer motivo es para ahorrar recursos y tiempo, ya que, en ese punto, se espera que no se pueda obtener un mejor rendimiento. La segunda razón surge del concepto de *overfitting*. Este fenómeno ocurre cuando el modelo se ajusta demasiado a los datos de entrenamiento, lo que le impide identificar patrones en ellos y, por lo tanto, generalizar cuando se le presentan nuevas entradas que no ha visto antes. Este concepto se puede entender mejor a través de una demostración gráfica, como la que se ve en la Ilustración 4.1.

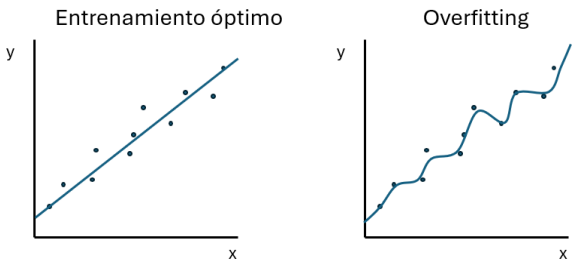

Ilustración 4.1: Demostración gráfica del *overfitting*

Para poder detectar el *overfitting* en los modelos, es necesario utilizar un conjunto de validación. Este conjunto se presenta a la red inmediatamente después de la actualización de pesos realizada en una época. Dicho conjunto está compuesto por datos que no han sido parte del entrenamiento, por lo que su capacidad para predecir correctamente las salidas esperadas determinará si se está logrando la capacidad de generalización o no.

El *overfitting* puede ser detectado cuando la gráfica de pérdidas obtenidas por la red neuronal durante su entrenamiento se asemeja a la mostrada en la Ilustración 4.2. En ella, se puede observar que, conforme van avanzando las épocas, pese a que el error en el conjunto de entrenamiento se va reduciendo, el error en el conjunto de validación incrementa significativamente a partir de una determinada época. Esto muestra que, como se expuso anteriormente, la red neuronal está comenzando a aprender de memoria los resultados que debe dar, y no los patrones generales de los datos de entrenamiento. Por tanto, un buen *Stopper* deberá finalizar el aprendizaje del modelo en ese momento.

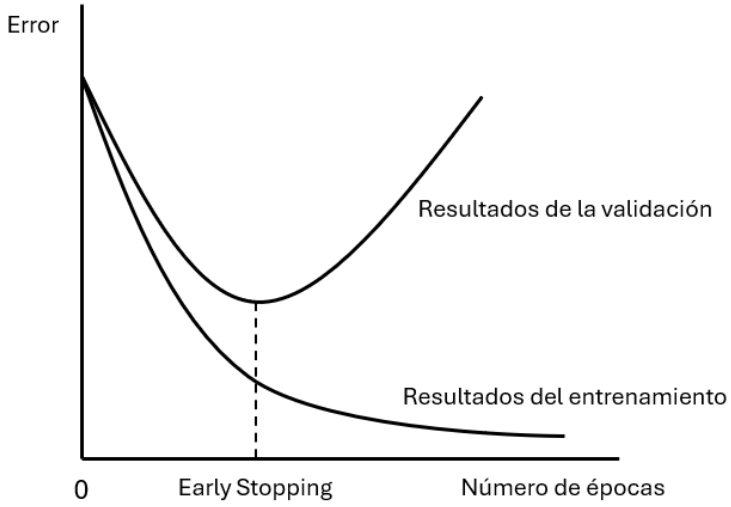

Ilustración 4.2: Gráfica de pérdidas de un modelo que muestra que esta sufriendo *overfitting*

4.1.6. Estrategias de entrenamiento

Dentro de la IA, las redes neuronales pueden ser entrenadas para muchos objetivos distintos. Por ejemplo, puedes crear un modelo que clasifique correctamente una serie de datos o usarlo para que genere nuevas imágenes. De esta decisión dependerá parte del entrenamiento del modelo, puesto que la verificación de los resultados se llevará a cabo de manera diferente según el objetivo seleccionado para la red. En este *framework*, esta decisión se llevará a cabo mediante el concepto de Estrategia. Esta será utilizada en todos los laboratorios para verificar el rendimiento del modelo entrenado en la tarea deseada por el usuario. Los resultados obtenidos de la estrategia de entrenamiento son esenciales para realizar comparaciones entre los diferentes modelos y decidir cuál tiene el mejor rendimiento.

Actualmente, se contemplan dos estrategias principales dentro *Flogo*. Por un lado, tenemos las clasificaciones, que permite a los usuarios entrenar modelos destinados a clasificar un objeto entre una serie de clases. En otras palabras, permite que las redes neuronales asignen una categoría específica a los datos de entrada en función de ciertas características. Para evaluar el rendimiento obtenido por el modelo una vez finalizado el entrenamiento, se comparan las etiquetas predichas con las esperadas.

Dentro de *Flogo*, se consideran dos estrategias fundamentales. Por un lado, las clasificaciones, que permiten a los usuarios entrenar modelos para asignar categorías específicas a los datos de entrada basándose en ciertas características. En otras palabras, estas estrategias posibilitan que las redes neuronales clasifiquen un objeto entre una serie de clases establecidas. Para evaluar el rendimiento del modelo una vez finalizado el entrenamiento, se comparan las etiquetas predichas con las esperadas.

Por otro lado, tenemos las regresiones, que permite crear modelos destinados a predecir nuevos valores. Las regresiones son técnicas estadísticas utilizadas para modelar la relación entre una variable dependiente y una o más variables independientes o predictoras. Su objetivo principal es predecir adecuadamente el valor de la variable dependiente en función de las variables predictoras.

4.1.7. Generador de datasets

Otro de los aspectos importantes dentro de este *framework* radica en la gestión de los *dataset*. Estos conjuntos de datos son los que se utilizarán en el entrenamiento del modelo de DL, tanto para mejorar su rendimiento como para la comprobación. Este aspecto resulta crítico para el resultado final, ya que sin una base de datos completa y variada, el modelo no será capaz de extraer las relaciones o patrones necesarios para aprender.

El proceso de creación de un *dataset* resulta ser muy complejo de generalizar por dos factores principales: existe una gran variedad de formas de representación y cada *framework* del mercado posee una manera distinta de cargarlos. Con respecto al primer factor mencionado, debemos tener en cuenta que, aparte de los tipos de datos que puedan existir, como números, imágenes, textos, audios, etc., hay diversas formas de definir cada uno de ellos. Por ejemplo, en un conjunto de datos de imágenes puede variar el formato de las imágenes, la

forma de expresar la categoría de cada una de ellas, su estructura de ficheros interna, entre otros aspectos. En cuanto al segundo factor, cada entorno de desarrollo posee su forma única de representar y cargar estas estructuras de datos. Particularmente, se puede observar cómo Pytorch solo trabaja con tensores, mientras que otros, como Keras, son mucho más flexibles en ese ámbito.

En resumen, resulta complicado lograr una implementación lo suficientemente general y abstracta que se adapte completamente a todas las posibilidades que puedan surgir. Por estas razones, se decidió establecer una forma común de representación y carga de *datasets* en nuestro *framework*, la cual se explicará más adelante en el sección de Implementación.

Dependiendo de los datos incluidos en el *dataset*, se pueden identificar varios tipos distintos. En la actualidad, *Flogo* ofrece la posibilidad de utilizar dos variaciones específicas de conjuntos de datos:

- ***Dataset* numérico:** Este se utiliza cuando el entrenamiento se lleva a cabo exclusivamente con valores numéricos. Son especialmente útiles para tareas como reconocimiento de imágenes, procesamiento de lenguaje natural, predicción, etc.
- ***Dataset* de imágenes:** Este tipo de *dataset* se refiere a aquellos compuestos por imágenes, donde cada imagen tiene asignada una etiqueta. Son usados principalmente en actividades como reconocimiento de objetos, clasificación de imágenes, segmentación semántica, etc.

4.1.8. Experimento

El concepto de experimento es fundamental dentro de todo el software desarrollado. Este se define como un conjunto de características o parámetros que determinarán cómo se llevará a cabo el procedimiento de aprendizaje de una arquitectura. Por lo tanto, el experimento es el objeto utilizado por el usuario para especificar cómo desea que se realicen los entrenamientos sobre una determinada red neuronal. El usuario tiene completa libertad para definir tantos experimentos como considere necesarios para alcanzar el rendimiento deseado.

Como indica su nombre, en esencia cada experimento no deja de ser un área de pruebas donde el desarrollador puede mezclar y probar diferentes configuraciones de hiperparámetros que considere que puedan obtener un resultado óptimo. Será el *framework* el encargado de probar cada una de estas variantes definidas y guardar cada uno de los modelos generados. Por tanto, la única función de cada experimento será ejecutar el entrenamiento de la red neuronal definida con los hiperparámetros establecidos.

En definitiva, los experimentos son un concepto clave dentro del *framework*, ya que permiten a los usuarios realizar todas las pruebas que deseen de forma ordenada y sistemática en una única iteración. Esto facilita a los desarrolladores la comparación rápida y clara de cada una de las ejecuciones, para determinar qué configuraciones de hiperparámetros benefician más a su objetivo final.

4.1.9. Laboratorio

En el ámbito experimental, los científicos acuden a un laboratorio para probar sus diferentes hipótesis a través de experimentos. Basándose en esto, un laboratorio puede definirse como un conjunto de experimentos encargado de gestionarlos. Se trata del concepto más general desarrollado y una de las bases del funcionamiento de todo el sistema.

El laboratorio delegan la tarea de entrenar la red a los experimentos. Su función se reduce a probar cada uno de los experimentos creados y almacenar todos los resultados obtenidos, tales como modelos entrenados o estadísticas del entrenamiento. Una vez completada la ejecución de todos los experimentos, selecciona aquel que haya generado un modelo con el mejor rendimiento y lo someten a prueba con el conjunto de datos de test, con el fin de obtener una evaluación real del rendimiento del modelo.

Como se comentó en la 1 y se verá posteriormente en la sección de Trabajo futuro, se tienen intenciones de continuar con el desarrollo de *Flogo*. En futuras iteraciones, los grandes objetivos serán alcanzar una heurística o procedimiento de mutaciones que permita al sistema averiguar, de forma automática, qué combinación de hiperparámetros es la más adecuada para el entrenamiento de la arquitectura dada. En todo este proceso, el laboratorio será una pieza fundamental, ya que será el encargado de gestionar y llevar a cabo este procedimiento. Por estas razones, el ejercicio de generalización llevado a cabo ha tenido grandes resultados, como la definición de este concepto.

4.2. Uso

Antes de profundizar en la explicación de cómo se ha implementado el *framework*, es esencial describir su utilización a través del DSL de *Flogo*. Para ello, se debe iniciar declarando un laboratorio, que supervisará el proceso de entrenamiento de una arquitectura, la cual estará presente en el mismo archivo, aunque será responsabilidad del *framework* estructural. Los pasos a seguir para la declaración del laboratorio son los siguientes:

4.2.1. Paso 1: Definición de los parámetros del laboratorio

El primer paso para obtener modelos entrenados es declarar el laboratorio que se utilizará, ya que este es el concepto más general y contiene al resto de componentes. Como se muestra en la Ilustración 4.3, se debe especificar el nombre asignado al laboratorio y el número de épocas que se desea ejecutar, siendo por defecto 1.

```
Laboratory(epochs = 10, name = "WineQuality")
```

Ilustración 4.3: Ejemplo de inicio creación de una instancia de un laboratorio a partir del DSL

A continuación, es necesario especificar todos los parámetros requeridos para la ejecución del laboratorio. Estos parámetros deben estar indentados y pueden ser añadidos sin un orden específico. Como se muestra en la Ilustración 4.4, se debe incluir un optimizador, una función de pérdidas, la estrategia de entrenamiento a seguir y, si el usuario lo desea, el *Stopper* a utilizar. Es importante destacar que los dos primeros serán definidos como valores por defecto en los experimentos. Esto significa que serán empleados en caso de que no se especifiquen en algún experimento en particular.

```
Laboratory(epochs = 10, name = "WineQuality")
    SGD(lr=0.001, momentum=0, momentumDecay=0, weightDecay=0)
    MSELoss
    RegressionStrategy
    LossDrivenEarlyStopper(10, 0.01)
```

Ilustración 4.4: Ejemplo de defínción de los parámetros de un laboratorio a partir del DSL

Por último, es necesario especificar el *dataset* que se utilizará. Para ello, se debe utilizar el objeto *Dataset* compatible con el DSL, manteniendo el mismo nivel de indentación utilizado anteriormente. A través de este objeto, se debe indicar el nombre del conjunto de datos que se cargará y el tamaño de *batch* usado cuando se vaya a entrenar la arquitectura, como se muestra en la Ilustración 4.5. Este concepto incluye un objeto *Split*, que definirá las proporciones utilizadas del *dataset* para los procesos de entrenamiento, prueba y validación del modelo.

```
Laboratory(epochs = 10, name = "WineQuality")
    SGD(lr=0.0001, momentum=0, momentumDecay=0, weightDecay=0)
    MSELoss
    RegressionStrategy
    LossDrivenEarlyStopper(10, 0.01)
    Dataset(name = "winequality-red", batchSize=10)
        Split(train = 0.7, test = 0.2, validation = 0.1)
```

Ilustración 4.5: Ejemplo de definición de un *dataset* a partir del DSL

4.2.2. Paso 2: Definición de los experimentos

El siguiente paso consiste en declarar todos y cada uno de los experimentos que se quieran probar dentro del laboratorio. Como estos no dejan de estar contenidos dentro del laboratorio, deberán mantener un nivel extra de indentación.

Para comenzar, es necesario utilizar la palabra reservada del DSL *Experiment*, seguida del nombre deseado. En el ejemplo presentado en la Ilustración 4.6, se emplea el nombre *b525*. Posteriormente, se agregarán todos los hiperparámetros que hagan único al experimento en creación. Actualmente, se pueden especificar funciones de pérdidas y optimizadores. En caso de que no se proporcionen, se emplearán los valores por defecto establecidos por el laboratorio. En el caso de la Ilustración 4.6, se observa que la función de pérdidas se sustituye, pero no se especifica ningún optimizador, lo que significa que se utilizará el valor predeterminado.

Además, otro parámetro que el desarrollador podrá emplear son los *Materialization*. Estos se usarán para asignar valores a las capas que, durante la definición de la red neuronal, se establecieron como parámetros. En otras palabras, permitirán probar el comportamiento de diferentes capas en puntos específicos indicados durante la creación de la arquitectura mediante el uso del objeto *VLayer*. Esta adición a *Flogo* fue diseñada con el propósito de disminuir la cantidad de arquitecturas declaradas, permitiendo que estas sean modificadas directamente desde el propio experimento. Por ejemplo, en el experimento de la Ilustración 4.6, vemos que para la *VLayer* con id 1 de nuestra arquitectura, especificada mediante el objeto *Materialization*, se le asignará una capa de normalización de *batch*.

```
Experiment b525
    Substitute(id="01") > BatchNormalization(eps=0.1, momentum=0.7)
    Substitute(id="02") > LogSigmoid
    Substitute(id="03") > Dropout(probability=0.6)
    MAELoss
```

Ilustración 4.6: Ejemplo de definción de un laboratorio completo a partir del DSL

Si se desea declarar más de un experimento, esto puede realizarse fácilmente mediante la declaración secuencial de cada uno, manteniendo la correcta indentación en cada nivel. En la Ilustración 4.7, se muestra cómo se vería la declaración de un laboratorio completo que incluye tres experimentos distintos, siguiendo las indicaciones proporcionadas.

```
Laboratory(epochs = 10, name = "WineQuality")
    SGD(lr=0.0001, momentum=0, momentumDecay=0, weightDecay=0)
    MSELoss
    RegressionStrategy
    LossDrivenEarlyStopper(10, 0.01)
    Dataset(name = "winequality-red", batchSize=10)
        Split(train = 0.7, test = 0.2, validation = 0.1)

    Experiment b525
        Materialization(vLayer="01") > BatchNormalization(eps=0.00001, momentum=0.3)
        Materialization(vLayer="02") > LogSigmoid
        Materialization(vLayer="03") > Dropout(probability=0.6)

    Experiment e626
        Materialization(vLayer="01") > BatchNormalization(eps=0.00001, momentum=0.3)
        Materialization(vLayer="02") > ReLU
        Materialization(vLayer="03") > Dropout(probability=0.5)

    Experiment a424
        Materialization(vLayer="01") > BatchNormalization(eps=0.00001, momentum=0.3)
        Materialization(vLayer="02") > Tanh
        Materialization(vLayer="03") > Dropout(probability=0.5)
```

Ilustración 4.7: Ejemplo de definción de un laboratorio completo a partir del DSL

4.2.3. Paso 3: Compilación

Una vez que se hayan declarado todos los objetos necesarios para gestionar el ciclo de vida de la red neuronal, procederemos a iniciar el proceso de compilación del modelo definido a través del DSL. Este proceso, que se explicará detalladamente más adelante en la sección de Implementación, generará los archivos Python necesarios para definir las arquitecturas especificadas con el *framework* estructural, así como el proceso de entrenamiento y almacenamiento de los modelos con el *framework* operacional.

Para lograr esto, se utilizará el generador de código de *Flogo* y las plantillas desarrolladas con ItRules. Una vez se tengan los archivos Python generados, simplemente se tendrá que ejecutar el archivo del laboratorio para probar cada uno de los experimentos especificados. También existe la opción de subir los archivos al servicio de laboratorios y probar estos experimentos a través de este servicio, que es explicado en el capítulo de servicio de laboratorio.

4.3. Implementación

Para dar soporte a todos los conceptos especificado anteriormente y permitir a los usuarios ejecutar el modelo especificado a través del DSL, se ha desarrollado un *framework* operacional que se encargará de ejecutar todos los procesos indicados.

4.3.1. Compilación

Un *framework* es una estructura conceptual y técnica que proporciona un conjunto de herramientas para desarrollar software de manera eficiente y consistente. Ofrece una base sólida y predefinida de funcionalidades y componentes comunes, lo que acelera el proceso de desarrollo al evitar la necesidad de repetir procedimientos. Los *frameworks* proporcionan un conjunto de clases base que son extendidas por cada implementación creada, ofreciendo una funcionalidad básica a todas las implementaciones. El *framework* operacional proporcionará a los desarrolladores un conjunto de herramientas abstractas a las tecnologías tradicionales de DL, que les permitirá administrar sus modelos de redes neuronales de manera más eficiente.

Las clases que implementen las interfaces del *framework* pueden ser generadas a partir del modelo especificado por un DSL. Para ello, se hará uso de un compilador, que se encargará de traducir el código escrito con *Flogo* a ficheros Python que usarán los *frameworks* creados, tanto el de arquitectura como este operacional.

El compilador hace uso del concepto *Template Based Code Generation* (TBCG) para generar los ficheros finales. La generación de código basada en plantillas es un enfoque de desarrollo de software que utiliza plantillas predefinidas para automatizar la creación de código. Estas plantillas contienen fragmentos de código con marcadores o variables que se llenan con datos específicos durante el proceso de generación. Al usar este método, se puede reducir la cantidad de código manual que necesitan escribir, mejorar la consistencia y la calidad

del código generado, y aumentar la productividad al eliminar tareas tediosas y propensas a errores. Las herramientas de generación de código basadas en plantillas pueden adaptarse a diferentes lenguajes de programación y entornos de desarrollo, lo que las hace versátiles y ampliamente utilizadas en diversas industrias. Para crear las plantillas que utilizará el compilador, se decidió emplear *ItRules*.

4.3.2. ItRules

ItRules [15] es un motor de plantillas basado en Java, capaz de generar código para el lenguaje de programación deseado a partir de una serie de reglas definidas. Estas reglas siguen la estructura *si x entonces y*, donde se distinguen dos partes principales: una condición (x) y una acción (y). Cada acción contiene el texto que se mostrará si se cumple la condición. Cuando esto ocurre, se dice que la regla se ha activado. A continuación, se muestra un ejemplo de una regla creada para este proyecto en la Ilustración 4.8, con el fin de explicar sus partes más importantes:

```
def type(architecture) and trigger(import)
    from $experiment_name+Lowercase import architecture as $experiment_name
end
```

Ilustración 4.8: Ejemplo de regla usada con *ItRules*

- **def:** Palabra restringida que marca el inicio de la definición de una regla.
- **type(), trigger():** Va seguido de *def*, y determina la condición que se debe cumplir para lanzar la regla. En el caso del*type()*, la regla se lanza cuando el objeto que la ejecuta es del tipo indicado entre paréntesis. En el caso del ejemplo, cuando es de tipo *architecture*. Para el *trigger()*, se comporta como lanzadores usados para diferenciar entre reglas con el mismo *type()*.
- **$:** Es un símbolo que indica el inicio de un marcador, siempre acompañado de una palabra que sirve para referirse a este. El valor que se asignará al marcador puede ser determinado por otra regla o mediante una sustitución directa desde el código.
- **+LowerCase:** Ejemplo de uso de formateador concreto. Estos permiten hacer modificaciones o transformaciones sobre el texto que se va a sustituir en los marcadores. En este caso, este formateador pone todas las letras a minúsculas.
- **end:** Palabra usada para indicar el final de una regla.

Una vez expuestas las reglas y sus componentes principales, es necesario explicar cómo se utilizan para generar el código requerido. En el caso de *ItRules*, el motor de plantillas utiliza un objeto para ejecutar las distintas reglas, las cuales se activan cuando todas sus condiciones son verdaderas. Si existen múltiples reglas aplicables, se ejecutará la que se encuentre en primer lugar en el código, ya que estas pueden ser ordenadas.

La característica principal de ItRules que lo distingue de otros motores de plantillas es su enfoque en la programación funcional. Las plantillas no contienen sentencias de control; en

su lugar, están compuestas por un conjunto de reglas que se ejecutan en función de premisas establecidas. Esto resulta en un código de plantilla más limpio, lo que evita errores y facilita el mantenimiento. Además, simplifica el proceso de análisis al reducir la complejidad sintáctica.

4.3.3. Reglas usadas

Las plantillas son fundamentales para generar el código necesario a partir del modelo especificado por el usuario en el DSL. Por ello, es importante explicar las reglas más significativas en este proceso. Estas reglas tienen como objetivo obtener el código necesario para ejecutar las acciones requeridas en el *framework*.

En la Ilustración 4.9, se puede observar la regla principal dentro de la plantilla creada. Esta regla contiene el esqueleto básico necesario para la generación del código adecuado del *framework*. Debe ser siempre la primera en ejecutarse, ya que incluye todos los marcadores de posición necesarios para personalizar el resultado y adaptarlo a los intereses del usuario.

```
def type(main)
    $architecture+import...[$NL]
    from framework.toolbox.laboratory import Laboratory
    from framework.toolbox.logger import Logger
    from framework.toolbox.stopper import EarlyStopper
    from implementations.$library.toolbox.experiment import $library+FirstUppercase~Device as Device
    from implementations.$library.toolbox.experiment import $library+FirstUppercase~Experiment as Experiment
    from implementations.$library.toolbox.data.generator import $library+FirstUppercase~DatasetGenerator as DatasetGenerator
    $optimizer+import...[$NL]
    $loss+import...[$NL]
    from implementations.$library.toolbox.saver import $library+FirstUppercase~ModelSaver as ModelSaver
    from implementations.$library.toolbox.loader import $library+FirstUppercase~ModelLoader as ModelLoader
    $strategy+import

    dataset = $dataset

    experiments = [$experiment...[, $NL$TAB$TAB$TAB$TAB]]

    $laboratory
end
```

Ilustración 4.9: Regla inicial para la generación de código del *framework*

El primer aspecto a comentar es cómo se genera la lista de *imports* necesarios para ejecutar el entrenamiento de la red neuronal, asegurando que solo se importen las clases que realmente se utilizarán a lo largo del programa. Cabe destacar que el uso de *as* se emplea para abstraer el *framework* subyacente en el código de ejecución. Esto permite crear un programa más legible y menos dependiente de la tecnología empleada.

Los primeros *imports* corresponden a aquellos que siempre se generarán de la misma manera, independientemente de la información que el usuario haya proporcionado a través del DSL. Estos importan clases que son de uso general y que no dependen del *framework* específico utilizado en la implementación. En la Ilustración 4.10, se pueden ver como se lleva a cabo la importación estática sin ningún tipo de marcas de texto.

```
from framework.toolbox.laboratory import Laboratory
from framework.toolbox.logger import Logger
from framework.toolbox.stopper import EarlyStopper
```

Ilustración 4.10: *Imports* dentro de la regla *main* que no se verán modificados

Posteriormente, encontramos un segundo bloque de líneas destinadas a agregar los *imports* de aquellas clases que siempre se utilizan, pero que requieren especificar el *framework* deseado de entre los implementados por *Flogo*. Por ello, en la regla mostrada en la Ilustración 4.11, se observa el marcador *$library*, que adquirirá el valor del entorno indicado, ya sea Pytorch, TensorFlow, Keras, u otro.

```
from implementations.$library.toolbox.experiment import $library+FirstUppercase~Device as Device
from implementations.$library.toolbox.experiment import $library+FirstUppercase~Experiment as Experiment
from implementations.$library.toolbox.data.generator import $library+FirstUppercase~DatasetGenerator as DatasetGenerator
from implementations.$library.toolbox.saver import $library+FirstUppercase~ModelSaver as ModelSaver
from implementations.$library.toolbox.loader import $library+FirstUppercase~ModelLoader as ModelLoader
```

Ilustración 4.11: *Imports* dentro de la regla *main* donde solo varía la implementación usada

El último bloque de importaciones está destinado a todas aquellas clases que dependen del modelo especificado por el usuario en el DSL. Dado que estos procesos son más complejos que los vistos anteriormente, estos *imports* tendrán sus propias reglas que determinan el código final. Como se puede observar en la Ilustración 4.12, cada línea busca activar la regla que tenga un *trigger(import)* y un *type* con el mismo nombre del marcador como condiciones. El uso de puntos suspensivos indica que esta regla se activará tantas veces como sea indicado desde el motor de plantillas.

```
$architecture+import...[$NL]
$optimizer+import...[$NL]
$loss+import...[$NL]
$strategy+import
```

Ilustración 4.12: *Imports* dentro de la regla *main* que varían en función del modelo especificado por el DSL

Los objetos cuyas importaciones dependen de la información proporcionada por el motor de plantillas son las arquitecturas que se entrenarán, los optimizadores y las funciones de pérdidas que el usuario elija probar, así como la estrategia a aplicar con los datos. Las reglas que añaden las líneas correspondientes a estos objetos, en el mismo orden mencionado, se reflejan en la Ilustración 4.13. Es importante destacar que todas estas reglas siguen una estructura común; las cuatro contienen dos marcadores. Uno de ellos (*$library*) se utiliza para indicar qué *framework* se desea usar de entre los implementados, mientras que *$name* se utiliza para especificar el nombre concreto del objeto que se quiere usar dentro de las opciones proporcionadas por *Flogo*.

```
def type(architecture) and trigger(import)
    from $name+Lowercase import architecture as $name
end

def type(optimizer) and trigger(import)
    from implementations.$library.toolbox.optimizers.$name+LowerCase \
        import $library+FirstUppercase$name+FirstUppercase~Optimizer as $name~Optimizer
end

def type(loss) and trigger(import)
    from implementations.$library.toolbox.losses.$name+LowerCase \
        import $library+FirstUppercase$name+FirstUppercase~LossFunction as $name~LossFunction
end

def type(strategy) and trigger(import)
    from implementations.$library.toolbox.strategies.$name+LowerCase \
        import $library+FirstUppercase$name+FirstUppercase~Strategy as $name~Strategy
end
```

Ilustración 4.13: Reglas usadas para añadir las líneas de *imports* variables

En este punto, ya hemos importado todos los objetos necesarios para ejecutar el entrenamiento especificado. Por lo tanto, continuaremos explicando las reglas utilizadas para generar el código que creará las instancias de cada objeto necesario. La primera regla será utilizada para generar el *dataset*, mostrada en la Ilustración 4.14. En esta regla, el motor de plantillas asignará los valores apropiados a cada uno de los parámetros necesarios mediante los marcadores de posición.

```
def type(dataset)
    DatasetGenerator(name="$datasetName",
                             path="$path",
                             batch_size=$batchSize,
                             random_state=$seed).generate(train_proportion=$trainProportion,
                                                          validation_proportion=$valProportion,
                                                          test_proportion=$testProportion)
end
```

Ilustración 4.14: Reglas usadas para generar el código del *dataset*

Para la generación de todos los experimentos especificados, contamos con dos reglas distintas, como se muestra en la Ilustración 4.15. La diferencia entre ambas radica en la inclusión del atributo *early_stopper*, ya que debemos considerar que este parámetro no es obligatorio y no debe ser siempre indicado. Los demás parámetros son agregados nuevamente a través de marcadores o mediante la activación de otras reglas secundarias.

```
def type(experiment) and attribute(early_stopper)
    Experiment(name="$experiment_name",
                                architecture=$architecture_name,
                                optimizer=$optimizer,
                                loss_function=$loss,
                                stopper=$early_stopper,
                                saver=ModelSaver("$saver_path"))
end

def type(experiment)
    Experiment(name="$experiment_name",
                                architecture=$architecture_name,
                                optimizer=$optimizer,
                                loss_function=$loss,
                                saver=ModelSaver("$saver_path"))
end
```

Ilustración 4.15: Reglas usadas para generar el código de los experimentos

Para concluir, debemos mencionar la última regla importante utilizada para personalizar el código utilizado en la creación de un laboratorio. Como se puede observar en la Ilustración 4.16, no existen diferencias relevantes con las reglas ya explicadas, ya que se utilizan marcadores para activar otras reglas, como la de definición de la estrategia, o para asignar valores directos, como las épocas.

```
def type(laboratory)
    Laboratory(name="$laboratoryName",
                eras=$eras,
                epochs=$epochs,
                datagen=dataset,
                experiments=experiments,
                strategy=$strategy,
                logger=Logger("$logger_path"),
                loader=ModelLoader(),
                device=Device($device)).explore()
end
```

Ilustración 4.16: Reglas usadas para generar el código de laboratorio

4.3.4. Código generado

Tan importante como el uso del *framework* a través del DSL es el código generado al final del proceso de compilación. Este código permite ejecutar todas las acciones requeridas por el usuario en Python. Por lo tanto, también es esencial revisar los resultados de la compilación y comprender los archivos Python generados. Para una explicación detallada, analizaremos el código generado a partir de la compilación del ejemplo presentado en la sección de Uso usando las reglas vistas.

Siguiendo la Ilustración 4.9, donde se muestra la regla principal creada, las primeras

líneas de los archivos Python creados corresponderán a todos los *imports* de objetos que sean necesarios a lo largo del programa

El próximo objeto que se generará será el encargado de cargar el *dataset* que se vaya a usar, el *DatasetGenerator*. Esto se llevará a cabo mediante la activación de la regla mostrada en la Ilustración 4.14. En esta regla, el compilador utilizará los valores indicados en el DSL para sustituirlos en las marcas de texto correspondientes. El resultado, como se muestra en la Ilustración 4.17, permitirá cargar el *dataset* que será utilizado en el *framework* operacional para cargar la red neuronal.

```
dataset = DatasetGenerator(name="winequality-red",
                           path="/api/flogo/execute/files/dataset/",
                           batch_size=10,
                           random_state=728).generate(train_proportion=0.7,
                                                      validation_proportion=0.2,
                                                      test_proportion=0.1)
```

Ilustración 4.17: Generación en Python del *dataset* especificado en el DSL

El objeto *DatasetGenerator* será responsable de cargar los datos destinados al entrenamiento del modelo. Por lo tanto, es importante en este punto explicar cómo se espera que el *framework* reciba los datos para que puedan ser cargados correctamente.

Para cargar un *dataset*, se ha optado por el uso de un archivo de metadatos con una estructura de clave-valor que debe ser creado por el usuario. Este formato se ha seleccionado por la flexibilidad que proporciona al desarrollador para especificar varios parámetros que determinen el proceso de creación del *dataset*. Por ejemplo, si el usuario desea cargar un archivo separado por comas (CSV), tiene la opción, pero no la obligación, de indicar parámetros básicos como el separador, el tipo de datos, etc. El archivo de metadatos mencionado tendrá un formato llamado TSV o *Tab Separated Values*. En este archivo se proporcionará toda la información necesaria para la correcta carga de los datos. El archivo siempre llevará el nombre de *meta-dataset.tsv* y deberá contener una serie de claves obligatorias que, en su mayoría, son comunes a todos los tipos de conjuntos de datos. Estas claves son:

- ***dataset:*** Es usado para indicar el tipo de *dataset* que se quiere cargar. Actualmente, hay dos variantes implementadas, que son el numérico (*numeric*) y el de imágenes (*images*).
- ***name:*** Debe estar acompañado del nombre que se le vaya a dar al *dataset* que se está subiendo.
- ***task:*** Indica la tarea para la que está diseñado el conjunto de datos. Como se explicó anteriormente, tenemos dos posibilidades: *regression* o *classification*.

El método *generate()*, también visible en la Ilustración 4.17, será el encargado de crear los tres *datasets* necesarios con los datos que han sido cargados previamente. Por esta razón, es importante indicar mediante parámetros en esta función las proporciones de datos que se destinarán a los conjuntos de entrenamiento, de prueba y de validación.

Una vez que se generado el proceso de carga del *dataset*, será necesario crear el código requerido para especificar todos los experimentos necesarios. En este caso, se utilizará la primera regla visible en la Ilustración 4.15, ya que se ha indicado que se va a utilizar un *Stopper*. Esta regla será invocada tres veces, dado que se han declarado tres experimentos distintos en el DSL. Todos estos experimentos serán almacenados dentro de una lista para que puedan ser posteriormente pasados al laboratorio. El código generado al final de este proceso se puede ver en la Ilustración 4.18.

```
experiments = [Experiment(name="b525",
                          architecture=b525,
                          optimizer=SGDOptimizer(parameters=b525.parameters(), learning_rate=0.001, momentum=0,
                                                 dampening=0, weight_decay=0.0),
                          loss_function=MAELossFunction(),
                          stopper=EarlyStopper(patience=10, delta=0.01),
                          saver=ModelSaver("api/flogo/model/WineQuality")),
               Experiment(name="e626",
                          architecture=e626,
                          optimizer=SGDOptimizer(parameters=e626.parameters(), learning_rate=0.001, momentum=0,
                                                 dampening=0, weight_decay=0.0),
                          loss_function=MSELossFunction(),
                          stopper=EarlyStopper(patience=10, delta=0.01),
                          saver=ModelSaver("/api/flogo/model/WineQuality")),
               Experiment(name="a424",
                          architecture=a424,
                          optimizer=AdamOptimizer(parameters=a424.parameters(), learning_rate=0.0001, betas=(0.9, 0.999),
                                                  eps=1.0, weight_decay=0.0),
                          loss_function=MSELossFunction(),
                          stopper=EarlyStopper(patience=10, delta=0.01),
                          saver=ModelSaver("/api/flogo/model/WineQuality"))]
```

Ilustración 4.18: Generación en Python de los experimentos especificados en el DSL

La clase *Experiment* es fundamental para el correcto funcionamiento del *framework*, ya que contiene todos los hiperparámetros que hacen único el entrenamiento de una arquitectura. En la Ilustración 4.19, se puede observar un diagrama de clases descrito con *Unified Modeling Language* (UML) del objeto *Experiment*, donde se aprecian todos las estructuras que lo componen.

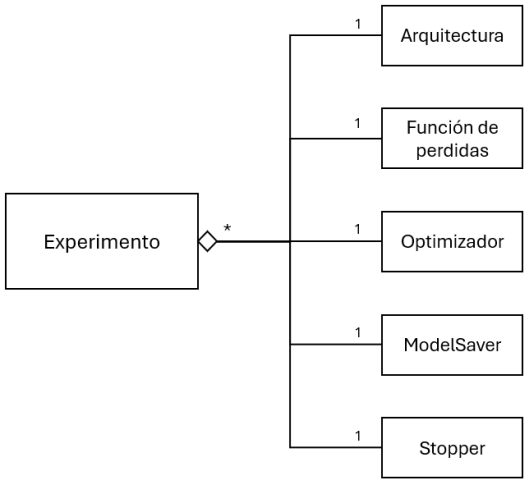

Ilustración 4.19: Diagrama de clases descrito con UML de un experimento

Finalmente, solo queda la generación de código para la creación de una instancia de un laboratorio. Este hará uso de la regla mostrada en la Ilustración 4.16 explicada anteriormente. Como se puede ver en el código generado, este recibirá la mayor parte de los objetos destinados a gestionar la ejecución de cada uno de los experimentos. El proceso de prueba de cada uno de los experimentos comenzará con la llamada a la función *explore()* que podemos ver en la Ilustración 4.20

```
Laboratory(name="WineQuality",
           eras=1,
           epochs=10,
           datagen=dataset,
           experiments=experiments,
           strategy=RegressionStrategy(MSELossFunction()),
           logger=Logger("/api/flogo/executions/logger/result.tsv"),
           loader=ModelLoader(),
           device=Device(-1)).explore()
```

Ilustración 4.20: Generación en Python del laboratorio especificados en el DSL

La clase *Laboratory*, como se describió anteriormente, será la encargada de gestionar la ejecución de todos los experimentos. Por lo tanto, como se muestra en la Ilustración 4.21, esta deberá recibir objetos que no afecten directamente a la ejecución de los experimentos, como el *Logger* o el *ModelLoader*, así como objetos que sean comunes a todos ellos, como el generador de *datasets*.

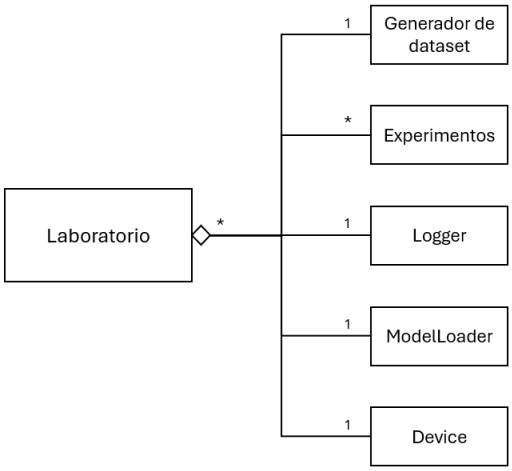

Ilustración 4.21: Diagrama de clases descrito con UML de un laboratorio

Todo el código generado a partir del compilador puede ejecutarse de dos formas distintas: directamente desde el archivo *.py* donde se almacene el laboratorio, o a través del servicio de laboratorio, que se explicará en el capítulo servicio de laboratorio. Aunque la segunda opción ofrece muchas ventajas al usuario, es importante destacar que el *framework* es completamente funcional sin el servicio.

Asimismo, a pesar de que el código haya sido explicado a partir del DSL, este no es fundamental para utilizar las clases proporcionadas por el *framework* operacional. Un usuario podrá, si lo necesita, hacer uso de ellas de forma directa, sin necesidad de utilizar el sistema completo de *Flogo*.

Capítulo 5

Servicio de laboratorio

El servicio de laboratorio es un servicio que se encarga de gestionar todos los archivos de arquitectura y laboratorio generados por los usuarios a través de *Flogo*, así como los *datasets* que vayan a utilizar. Su principal objetivo es facilitar la gestión de todas las pruebas de entrenamiento de modelos que los desarrolladores deseen realizar con los *frameworks* que conforman *Flogo*. Para ello, proporciona a los usuarios un repositorio propio donde pueden almacenar todos los archivos generados. Además, este servicio permite ejecutar los laboratorios subidos mediante una sencilla llamada a su API, administrando todos los resultados obtenidos durante el entrenamiento de una arquitectura, incluidos los modelos y sus estadísticas.

Este servicio puede resultar muy útil, ya que alivia muchas responsabilidades del usuario. Por un lado, automatiza la administración de todos los archivos generados, eliminando la necesidad de hacerlo manualmente. Además, evita que los desarrolladores tengan que acceder manualmente a los resultados del entrenamiento y a los modelos obtenidos, ya que el servidor los proporciona a través de llamadas a la API. Esto hace de *Flogo* un sistema más simple y atractivo para personas con menos conocimientos sobre DL y redes neuronales. Además, el servicio mejora la eficiencia en los procesos de creación de modelos al ahorrar tiempo tanto en procedimientos repetitivos como en la interpretación de los resultados de los modelos.

Actualmente, el servicio de laboratorio debe ejecutarse localmente, es decir, en el mismo sistema desde el que se realicen las peticiones. Como trabajo futuro, se deberá lograr que este servicio pueda ser ejecutado desde un servidor, permitiendo así que se le puedan hacer peticiones de forma remota. Asimismo, es importante recalcar que este servicio es un añadido a *Flogo*, ya que el sistema puede funcionar sin necesidad de utilizarlo.

5.1. Recursos

Para la comunicación necesaria entre servicio y usuario, se hace uso de una API REST. Esta es un enfoque arquitectónico para diseñar servicios web que permite a diferentes sistemas comunicarse de manera eficiente a través de Internet. Este estilo de API se basa en la

representación uniforme de recursos, donde la información se transmite en formatos estándar como *JavaScript Object Notation* (JSON).

Las interacciones cliente-servidor se realizan mediante métodos HTTP como GET para obtener datos, POST para enviar datos nuevos, PUT para actualizar recursos existentes y DELETE para eliminarlos. Las utilidades que han sido dadas a cada una de estas operaciones la veremos posteriormente. Otra cualidad básica de las API REST es que son sin estado. En otras palabras, cada solicitud contiene toda la información necesaria para que el servidor procese la petición, permitiendo no depender de sesiones o solicitudes previas, logrando tener independencia entre las diferentes solicitudes.

Al utilizar una API REST, somos capaces de crear un servicio web que permita ver de forma clara y directa al usuario las diferentes acciones que puede ejecutar. Además, estos servicios son altamente escalables y fáciles de mantener, facilitando la integración de sistemas y posibles ampliaciones que se lleven a cabo en el futuro. Los recursos actuales, y sus respectivas acciones, son los siguientes:

- ***/flogo/architecture.*** Usado para interactuar con todo los ficheros Python que representan arquitecturas definidas con el *framework* estructural. Se pueden llevar a cabo las siguientes acciones:
  - **POST:** Entrada de la API usada para subir ficheros al servicio. Para realizar esta llamada correctamente, el cuerpo de la solicitud debe contener un objeto JSON en la sección *definition*, con el tipo de contenido *application/json*. Este objeto debe tener dos atributos principales: *name*, que corresponde al nombre dado por el usuario al objeto a cargar, y *content*, que contiene todo el contenido del archivo a subir.
  - **GET:** Esta entrada proporcionará a los usuarios la capacidad de solicitar un listado de los nombres de todos los archivos disponibles. En el campo *response* del JSON de respuesta, se encontrará un *array* con todos los archivos disponibles en el repositorio.
- ***/flogo/laboratory.*** Representa todos aquellos archivos Python que contengan un laboratorio definido con el *framework* operacional. Con estos, se pueden ejecutar ciertas acciones:
  - **POST:** *Endpoint* usado para colgar ficheros de laboratorio al servicio. Al igual que antes, se deberá añadir en el cuerpo de la *request* un objeto JSON que contenga el nombre del objeto y su contenido.
  - **GET:** Usada para solicitar al servicio un listado con todos los laboratorios disponibles. La respuesta se obtendrá en el campo *response* del JSON.
- ***/flogo/dataset.*** Son todos aquellos archivos que representen datasets que vayan a ser subidos al repositorio para ser usados posteriormente. Las acciones que permiten son:
  - **POST:** Llamada que indica que se va a subir un nuevo *dataset* al servicio. Además del mismo JSON que contenga el nombre asignado al conjunto de datos, es necesario agregar una sección adicional llamada *attachment* en el cuerpo de la solicitud,

donde se incluyen los bytes del archivo ZIP a cargar.

- **GET:** Como respuesta, se obtiene un listado de los *datasets* disponibles en el repositorio. Al igual que en los anteriores casos, se devuelve un *array* en el campo *response*.

- ***/flogo/architecture/:name.*** Este recurso es usado para indicar una arquitectura específica dentro de las disponibles en el repositorio. En el campo *name*, se debe incluir el nombre del objeto sobre el que se quiere hacer la acción. Las acciones que se pueden llevar a cabo son:
  - **GET:** El propósito de este *endpoint* es permitir a los usuarios solicitar la descarga de una arquitectura específica. El contenido del archivo estará disponible en el cuerpo de la respuesta recibida.
  - **DELETE:** La llamada a este *endpoint* iniciará el proceso de eliminación del archivo de arquitectura especificado mediante el parámetro *name*. La respuesta indicará si ha tenido lugar alguna incidencia durante la ejecución del proceso.
- ***/flogo/laboratory/:name.*** Recurso destinado a llevar a cabo una acción sobre algunos de los laboratorios que hayan sido subidos, indicando en *name*, en la laboratorio en cuestión. Las acciones que se pueden realizar son:
  - **GET:** Acción usada para descargar un archivo laboratorio específico de los que hayan sido subidos al repositorio. Al igual que antes, el archivo resultante podrá ser encontrado en el *body* de la respuesta.
  - **DELETE:** *Endpoint* que permite la eliminación de un laboratorio del repositorio del servicio. Si ocurre algún error durante el proceso, como no haber encontrado el archivo especificado, se indicará en la respuesta.
- ***/flogo/dataset/:name.*** Permite ejecutar una operación sobre alguno de los *datasets* disponibles. El campo *name*, como en el resto de casos, es usado para indicar el nombre específico del objeto. Las acciones disponibles en la API son:
  - **GET:** Permite descargar el *dataset* completo que posea el mismo nombre que el indicado por parámetros dentro del repositorio. Dentro del cuerpo de la respuesta, se podrán encontrar los bytes del archivo ZIP del conjunto de datos solicitado.
  - **DELETE:** Entrada usada para eliminar alguno de los *datasets* que haya sido subido previamente. A través de la respuesta, se sabrá si se ha finalizado el proceso correctamente.
- ***/flogo/execute/:laboratory.*** Recurso usado para indicar que se quiere llevar a cabo la ejecución de un laboratorio determinado, indicado a través del parámetro *laboratory*. La única acción valida en este recurso es la siguiente:
  - **POST:** Este *endpoint* será invocado cuando se desee ejecutar alguno de los laboratorios previamente cargados. En la respuesta JSON se devolverá el experimento que haya tenido el mejor rendimiento entre todos los probados.

- ***/flogo/stats/:stat.*** Indica el deseo de obtener una estadística o *stat* de las disponibles dentro del servicio. Esta deberá ser indicada a través del parámetro *stat*. Además, como parámetros de consulta (*query*), se deberán indicar todos los valores propios de la *stat* que sean necesarios. La acción es disponiblee es la siguiente.
  - **POST:** Arrancará el proceso de obtención de la estadística indicada. La respuesta contendrá otro JSON que será el resultado de la estadística solicitada. Las *stats* disponibles actualmente serán explicadas en la sección de implementación
- ***/flogo/model/:laboratory/:experiment.*** Entrada a la API usada para indicar que se quiere obtener los ficheros necesarios para ejecutar un contenedor con alguno de los modelos entrenados. Para ello, se deberá indicar el laboratorio y el experimento del que se quiera obtener el modelo. Su única acción se describe a continuación:
  - **GET:** Esta última acción deberá ser invocado cuando se desee obtener uno de los modelos entrenados con el servicio de laboratorio. Al igual que en llamadas anteriores, la respuesta contendrá el archivo ZIP solicitado en su cuerpo (*body*).

Los *endpoints description* de la API proporcionan una amplia gama de opciones para que los usuarios indiquen las operaciones que desean realizar. Las respuestas siempre serán un objeto JSON con un atributo *response*, donde se indicará el mensaje. Las entradas disponibles actualmente son las siguientes:

5.2. Uso

Para poder utilizar el servicio de laboratorio, se deben cumplir dos condiciones. La primera es que el servicio esté corriendo en la máquina local. La segunda es disponer de archivos de arquitectura, laboratorios y *dataset*, necesarios para probar su funcionamiento, ya que sin estos no existe ninguna funcionalidad útil. Cumplidos estos dos requisitos, se podrá usar el servicio.

Como se mencionó en la sección de recursos, los *endpoints* que ofrece la API desarrollada pueden resultar complejos, especialmente aquellas llamadas que requieren añadir un fichero al cuerpo de la solicitud. Con el objetivo de facilitar el trabajo al usuario, se ha desarrollado una interfaz que automatiza todas las llamadas posibles a la API. De esta forma, el uso del servicio será mucho más sencillo y menos complejo.

Por tanto, en este apartado se explicarán las posibilidades de esta interfaz y cómo se utiliza cada una de ellas, ya que es la forma más sencilla de usar el servicio. No obstante, la API puede ser llamada de forma manual y directa siempre que el desarrollador lo considere necesario; el uso de la interfaz no es obligatorio.

5.2.1. Subida de un objeto

Este método será utilizado por el usuario siempre que se quiera subir un objeto, ya sea una arquitectura, un laboratorio o un *dataset*, ya que simplifica el proceso de subida de archivos al repositorio a través de la API. Esto resulta realmente útil, puesto que añadir un fichero, sin importar su extensión, al cuerpo de una *request* puede ser un procedimiento complejo. La interfaz se encarga de este proceso, eliminando esta responsabilidad del usuario.

Para realizar esta operación, se debe indicar a la interfaz tres parámetros. El primero es la palabra *post*, que está reservada para esta operación. El segundo parámetro especifica el tipo de objeto que se desea subir, pudiendo ser una arquitectura, un laboratorio o *dataset*. El tercero indica la ruta donde se encuentra el archivo que se quiere subir. Cabe destacar que, en el caso de los *datasets*, se debe proporcionar el directorio que contiene todos los archivos necesarios para su declaración en el *framework* operacional, tal como se explicó en la sección de implementación.

En la Ilustración 5.1 se pueden ver tres llamadas distintas y sus correspondientes respuestas. Se debe sobre entender que los ficheros *linear_b525.py* y *wine_quality.py* son algunos de los ficheros de arquitectura y laboratorio respectivamente que fueron generados anteriormente. Además, es importante destacar que la interfaz no muestra el JSON completo de la respuesta, sino que se limita a mostrar el mensaje.

```
java -jar api-interface.jar post architecture /home/flogo/generated_files/linear_b525.py
The object named b525 have been uploaded

java -jar api-interface.jar post laboratory /home/flogo/generated_files/wine_quality.py
The object named WineQuality have been uploaded

java -jar api-interface.jar post dataset /home/flogo/datasets/numeric_dataset
The object named winequality-red have been uploaded
```

Ilustración 5.1: Ejemplos de subidas de ficheros al servicio hechas con la interfaz

5.2.2. Obtención de un objeto

Esta función se encargará de solicitar y descargar el archivo que haya solicitado el usuario, almacenándolo en la ubicación proporcionada como parámetro. Será utilizada cuando el desarrollador desee recuperar alguno de los archivos subidos al servicio, una operación esencial en cualquier repositorio.

Para ejecutar esta operación correctamente, el cliente deberá suministrar varios parámetros. El primero debe ser *get*, que es la palabra reservada para esta operación. Posteriormente, deberá indicar qué tipo de objeto quiere descargar y su nombre a través del segundo y tercer parámetro, respectivamente. El último indicará la ruta donde se guardará el objeto descargado. En caso de que se quiera descargar un *dataset*, es importante que el fichero donde se guardará tenga extensión ZIP. En la Ilustración 5.2, se pueden ver ejemplos de esta llamada.

```
java -jar api-interface.jar get architecture b525 /home/flogo/generated_files/architecture.py
File downloaded

java -jar api-interface.jar get dataset winequality-red /home/flogo/datasets/dataset.zip
File downloaded
```

Ilustración 5.2: Ejemplos de descargas de ficheros del repositorio hechas con la interfaz

5.2.3. Listado de objetos disponibles

Obtener un listado de archivos que han sido subidos al repositorio es crucial para conocer el estado del almacén de ficheros en un momento determinado. Para llevar a cabo esta tarea, contamos con este método de la interfaz. Su respuesta mostrará el nombre de todos aquellos objetos del tipo indicado que se encuentren en el repositorio.

Para la ejecución de este método, solo serán necesarios dos parámetros. El primero de ellos, como en todos los casos, se usará para indicar la palabra reservada de la operación que se quiera realizar. En este caso, la palabra es *list*. El segundo parámetro se utilizará para indicar el tipo del objeto del que se desea obtener el listado. En la Ilustración 5.3, se pueden ver dos ejemplos de usos del listado. Como se observa, en caso de que haya más de un objeto, se mostrará el nombre de cada uno de ellos en una línea separada.

```
java -jar api-interface.jar list architecture
b525
e626
a424

java -jar api-interface.jar list laboratory
WineQuality
```

Ilustración 5.3: Ejemplos de listados de ficheros en el repositorio hechos con la interfaz

5.2.4. Eliminación de un objeto

Esta función realiza la última de las operaciones principales dentro de un repositorio, ya que permite eliminar archivos que han sido subidos. Esto es esencial para que el usuario pueda tener un control completo de todos sus archivos dentro del almacén.

Para su correcta ejecución, solo son necesarios 3 parámetros sencillos. La palabra reservada para este método, que es *delete*, seguida del tipo de objeto que se va a eliminar y su nombre, siendo indicados en este orden. En este caso, debido a la simplicidad en la llamada del método y a la homogeneidad de este para todos los tipos de objetos, en la Ilustración 5.4 solo se ha mostrado una llamada.

```
java -jar api-interface-.jar delete architecture a424
File delete
```

Ilustración 5.4: Ejemplo de eliminación de un fichero del repositorio hecho con la interfaz

5.2.5. Ejecución de un laboratorio

Cuando se quiera llevar a cabo el entrenamiento de alguna de las arquitecturas que haya sido subida, será invocado este método. Cuando esto suceda, se ejecutará el laboratorio indicado. Esta función es la más importante de todas a las que da soporte la interfaz, ya que permite ejecutar el entrenamiento de una red neuronal de una forma muy sencilla.

La llamada de ejecución tiene solo dos parámetros. El primero es común en todas las funciones, utilizado para indicar qué operación se desea llevar a cabo mediante una palabra reservada. En este caso, es *execute*. Posteriormente, solo será necesario indicar el laboratorio que se quiera ejecutar, y comenzará el proceso de entrenamiento. Como se puede ver en la Ilustración 5.5, en la respuesta se indicará el nombre del experimento que mejor rendimiento haya tenido.

```
java -jar api-interface.jar execute WineQuality
The architecture with the best performance was b525
```

Ilustración 5.5: Ejemplo de ejecución de un laboratorio hecho con la interfaz

5.2.6. Obtención de una stat

Este método permite a los usuarios obtener diversas estadísticas o *stats* proporcionadas por el servicio, que muestran cómo han transcurrido las ejecuciones de los laboratorios. Estas estadísticas son fundamentales ya que ofrecen información relevante sobre los experimentos probados y, por ende, sobre los conjuntos de hiperparámetros utilizados.

Para invocar cada una de las *stats*, se deben proporcionar al menos dos parámetros. Estos son la palabra reservada para esta operación, que es *stat*, y la estadística que se quiere obtener de las disponibles. Las estadísticas disponibles pueden ser revisadas en la sección de implementación. Dependiendo de la *stat* que se quiera obtener, se deberá indicar una serie de parámetros adicionales, como se muestra en la Ilustración 5.6.

```
java -jar api-interface-.jar stat best-experiment laboratory=WineQuality
{"laboratory":"WineQuality","experiment":"b525","measurement":"25.558797955513"}

java -jar api-interface-.jar stat experiments-on-laboratory laboratory=WineQuality
{"laboratory":"WineQuality","experiments":["b525", "e626", "a424"],"measurements":["24.97614625784067",
"27.766752637", "26.1265267394"]}

java -jar api-interface-.jar stat experiment-stat laboratory=WineQuality experiment=b525
{"laboratory":"WineQuality","experiment":"b525","loss":["31.18070778479943","30.6373898432805",
"31.067030833317684","29.468221811147835","29.05016855093149","28.475565836979793","26.454752115102913",
"26.517045974731445","24.97614625784067","25.136541073138897"]}
```

Ilustración 5.6: Ejemplos de llamadas a *stats* disponibles en el servicio hechas con la interfaz

5.2.7. Obtención de un modelo

Los usuarios deben tener la posibilidad de obtener aquellos modelos que hayan sido entrenados. Por ello, esta función se encargará de proporcionar un archivo ZIP a los usuarios que contiene todos los archivos necesarios para ejecutar una imagen de Docker con el modelo indicado. Con esta, los desarrolladores podrán incorporar sus redes neuronales entrenadas a sus aplicaciones o darles el uso que consideren apropiado.

Son necesarios cuatro parámetros: el primero será *model*, que es la palabra reservada para esta operación. El segundo y el tercero serán el laboratorio y el experimento del que se quiera obtener el modelo, respectivamente. El último parámetro se utilizará para indicar la ruta donde se va a guardar el fichero comprimido descargado. En la Ilustración 5.7, se puede ver un ejemplo de una llamada de este tipo.

```
java -jar api-interface.jar model WineQuality b525 /home/flogo/models/b525.zip
File downloaded
```

Ilustración 5.7: Ejemplo de obtención de un modelo hecho con la interfaz

5.3. Implementación

Una vez explicado cómo se utiliza el servicio, es necesario abordar los conceptos más básicos de su implementación. Se repasarán los aspectos más importantes y que merecen mención, evitando detalles específicos del código que lo soporta.

5.3.1. Sistema de ficheros

Una de las principales funcionalidades de este servicio de laboratorio será proporcionar a los usuarios un repositorio donde puedan almacenar sus archivos, tanto de las arquitecturas y laboratorios desarrollados con *Flogo* como de los *datasets* necesarios para su ejecución.

Asimismo, la máquina donde se ejecute el servicio también almacenará todos los resultados obtenidos durante los procesos de entrenamiento, validación y test.

Por tanto, es fundamental explicar la estructura interna del sistema de archivos utilizado para almacenar todos estos archivos de manera ordenada y clara. Esta estructura se puede observar en el esquema de la Ilustración 5.8. Cabe destacar que todos los directorios mencionados a continuación se basan en una ruta inicial proporcionada por el usuario al iniciar el servicio.

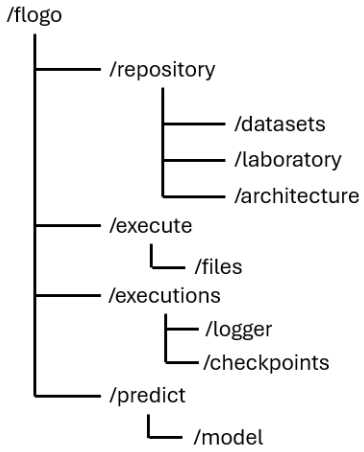

Ilustración 5.8: Estructura de ficheros interna del servidor que da soporte a la *API*

El **directorio repositorio** es la carpeta donde se almacenarán todas las arquitecturas, laboratorios y *datasets* subidos. Este se encuentra en la ruta */flogo/repository*, partiendo de la ruta inicial mencionada anteriormente. Internamente, se organizan tres subdirectorios, cada uno destinado a uno de los tipos de objetos que pueden ser subidos, con la siguiente información.

- **/flogo/repository/architecture/:** Este subdirectorio almacenará todas las arquitecturas subidas por el usuario. Habrá un archivo por cada arquitectura, nombrado de acuerdo con el mismo objeto. Además, existirá un archivo TSV que relacionará cada nombre de archivo con su respectivo resumen MD5, lo que permitirá verificar si un nuevo archivo ya existe en el servidor.
- **/flogo/repository/laboratory/:** Este directorio está destinado a almacenar todos los archivos *.py* que definan laboratorios. Similarmente, habrá un archivo por cada laboratorio subido, nombrado según lo especificado por el usuario. También existirá un archivo TSV que relaciona cada laboratorio con su correspondiente resumen MD5.
- **/flogo/repository/dataset/:** Este directorio contendrá todos los *datasets* subidos por el usuario, los cuales estarán en formato *.zip*. Estos archivos incluirán los datos para entrenar la red neuronal y los archivos necesarios para definir el conjunto en el *framework*. Similar a los apartados anteriores, existirá un archivo *dataset.tsv* destinado a evitar que se suban *datasets* duplicados.

El **directorio de ejecuciones** será una de las carpetas fundamentales del servidor, ya que es donde se llevarán a cabo las ejecuciones de los laboratorios. Estará ubicado en la ruta *flogo/execute/* y contendrá todos los archivos necesarios para realizar las ejecuciones de entrenamiento de las redes neuronales solicitadas por el usuario. Por lo tanto, podemos distinguir dos tipos principales de archivos dentro de este directorio: los estáticos y los dinámicos.

Los archivos estáticos son aquellos que se utilizan de la misma manera en todas las ejecuciones de laboratorios. Por lo tanto, no es necesario crearlos ni eliminarlos antes o después de comenzar su ejecución. Entre estos tipos de archivos se encuentran el entorno virtual de Python necesario para ejecutar código en este lenguaje y el archivo *requirements.txt*, que indica las librerías necesarias para su ejecución.

En cuanto a los archivos dinámicos, son aquellos que deben variar según el experimento solicitado para la ejecución. Por lo tanto, estos archivos deben ser movidos a este directorio desde su origen cuando se quieran utilizar, y luego eliminados cuando finalice el proceso de ejecución. Entre estos archivos se incluyen las arquitecturas que se desean entrenar, el laboratorio a ejecutar y el *dataset* descomprimido a utilizar. Todos estos archivos se almacenarán en un directorio adicional y temporal llamado *files*, creado para facilitar la eliminación posterior de los archivos temporales.

El **directorio de resultados**, situado en */flogo/executions/*, tiene como principal objetivo almacenar en archivos todos los resultados, medidas y modelos o *checkpoints* generados durante las ejecuciones de los laboratorios. Internamente, se pueden encontrar dos subdirectorios más:

- **/flogo/executions/logger/:** En este directorio se guardará el archivo *result.tsv*, donde los procesos de entrenamiento, validación y prueba volcarán todos los resultados obtenidos durante su ejecución a través del *Logger*. En otras palabras, este archivo de registro nos proporcionará información sobre la evolución de la red neuronal a lo largo de las diferentes épocas.
- **/flogo/executions/checkpoints/:** Ruta seleccionada para almacenar todos los *checkpoints* o modelos entrenados obtenidos. Este se encuentra organizado por laboratorios, existiendo una carpeta para cada uno de ellos. Dentro, se podrán observar los experimentos que lo componen, que contendrán todos los *checkpoints* obtenidos.

El **directorio de predicciones** se encuentra en la ruta */flogo/predict/*. Este directorio está destinado a contener todos los archivos necesarios para proporcionar al usuario la posibilidad de ejecutar los modelos entrenados. Esto se llevará a cabo a través de la tecnología de Docker, devolviendo al usuario todos los ficheros para lanzar su propio contenedor, al cual se le podrán hacer consultas que serán resueltas por el modelo entrenado.

Para lograr esto, se requieren varios archivos. El principal es el *Dockerfile*, utilizado para construir automáticamente la imagen de Docker. Este archivo hará uso del *requirements.txt*, que especificará las librerías de Python a instalar dentro de la imagen para la correcta ejecución del modelo. Además, en este directorio también se encontrarán las clases de *Flogo*, necesarias para ser incluidas dentro de la imagen. Finalmente, en esta ruta estará el archivo *service.py*, que será ejecutado dentro del contenedor para gestionar cada una de las peticio-

nes. Todos estos elementos estarán contenidos dentro de una carpeta llamada *model*, que será comprimida y entregada al usuario para permitirles ejecutar los contenedores.

5.3.2. Operaciones

Después de haber explicado el sistema de archivos utilizado por el servicio, a continuación, se comentará el procedimiento seguido por el servicio para permitir cada una de las acciones y llamadas explicadas previamente.

Una de las operaciones básicas dentro del servicio es la que permite **subir un objeto al repositorio**. Cuando llega una solicitud de este tipo, el servicio de laboratorio sigue una serie de pasos comunes. Se inicia descargando el JSON pasado en la solicitud a la API, extrayendo su contenido y nombre elegido por el usuario para el objeto. Si se trata de un *dataset*, se descarga el archivo ZIP del cuerpo de la solicitud.

Luego, se realizan dos comprobaciones clave. Primero, se verifica la unicidad del nombre dentro del repositorio. Si ya existe, se notifica al cliente con un mensaje de error. Segundo, se comprueba si el contenido ya ha sido subido previamente utilizando su resumen MD5. Si se encuentra una coincidencia, se devuelve el nombre del archivo existente en el servidor. Superadas estas comprobaciones, el objeto se guarda en el repositorio, ya sea como archivo *.py* para arquitecturas y laboratorios, o como *.zip* para *datasets*. Finalmente, se actualiza el archivo índice con el nuevo nombre y resumen.

Otra acción que debemos mencionar es la de obtención de un **fichero del repositorio**. La gestión de este tipo de solicitudes es más simple en comparación con el método anterior. Se realiza una comprobación para verificar si el objeto con el nombre indicado está en el repositorio del servidor. Si es así, se entrega al usuario a través del cuerpo de la respuesta. En caso contrario, se devuelve un mensaje de error que explica la situación.

Listar los objetos disponibles es una operación muy importante dentro del servicio, pero bastante sencilla de ejecutar en comparación al resto. El proceso seguido para obtener los archivos disponibles en el repositorio es el mismo tanto para los *datasets* como para las arquitecturas y los laboratorios. Primero, se extrae el nombre de todos los archivos que hayan sido subidos al repositorio desde el archivo índice correspondiente al objeto solicitado. Cada nombre se añade al JSON de respuesta en forma de *array*. Si no hay ningún archivo subido, en lugar de devolver la serialización de un *array* vacío, se retorna un mensaje de error que indica la situación.

También es crucial permitir la **eliminación de archivos del repositorio**, y el servicio sigue un proceso específico para esta operación. Una vez obtenido el nombre del objeto que se desea eliminar, se verifica su existencia en el repositorio. Si no se encuentra, significa que se está intentando eliminar un objeto que no ha sido subido al servidor, por lo que se devuelve un mensaje de error al usuario indicando esta situación. En caso de existir, se procede con su eliminación. Posteriormente, se elimina la línea que contiene su nombre y su resumen MD5 del archivo índice correspondiente para, finalmente, enviar al usuario una respuesta indicando el éxito del procedimiento.

Una de las funciones más importante del servicio es la que permite **ejecutar un laboratorio**. Como en todas las situaciones anteriores, se sigue un procedimiento específico para gestionar estas peticiones de manera adecuada. Primero, se carga el contenido del laboratorio que se desea ejecutar. Si no existe, se informa al usuario que debe cargar el archivo antes de proceder con la ejecución. Si el laboratorio está disponible, se mueve al directorio */flogo/execute* para su ejecución. Luego, se extraen el nombre del *dataset* y las arquitecturas del contenido. Se repite este proceso para todos los objetos necesarios en la ejecución, asegurando que estén ubicados correctamente en el directorio de ejecuciones.

Una vez que todos los archivos necesarios están en su lugar, se ejecutará un *script* de Python utilizando el entorno virtual existente en el servidor para iniciar el proceso de entrenamiento de las arquitecturas especificadas. Durante la ejecución, los resultados y los *checkpoints* se almacenarán en las rutas correspondientes para su acceso futuro. Una vez que se completen los pasos del laboratorio, se eliminarán los archivos movidos previamente para evitar ocupar espacio innecesario en memoria y para garantizar que las futuras ejecuciones no se vean afectadas. Por último, se identificará la arquitectura con el mejor rendimiento a partir de los resultados almacenados en el *Logger*, y se proporcionará su nombre al desarrollador en la respuesta.

El servicio también ofrece la capacidad de **obtener estadísticas** que reflejen el progreso del entrenamiento de una red neuronal. Actualmente, se admiten tres métricas diferentes, aunque existe la posibilidad de ampliar esta lista en el futuro. Todas las estadísticas son obtenidas a partir de los resultados guardado en el fichero de *log*. Cada una de estas métricas tiene una utilidad específica:

- ***best-experiment:*** Esta es la *stat* más sencilla de todas. Dado un laboratorio, se devuelve el nombre del experimento que haya obtenido los mejores resultados durante su ejecución. Esta información es especialmente valiosa para identificar qué combinaciones de hiperparámetros han sido más efectivas. Además del nombre del experimento, la respuesta incluirá el nombre del laboratorio y la medida obtenida en el conjunto de test.

- ***experiments-on-laboratory:*** Este método es similar al anterior, pero proporciona una visión más detallada de los experimentos realizados en un laboratorio específico. Devuelve todos los experimentos ejecutados en el laboratorio indicado, junto con sus valores de pérdida más bajos obtenidos durante el entrenamiento. Esta métrica es útil para comprender el rendimiento general de todos los experimentos y las diferencias reales entre ellos. La respuesta será un JSON que incluirá el laboratorio indicado, todos los experimentos realizados y sus pérdidas más bajas.

- ***experiment-stat:*** Esta estadística proporciona información detallada sobre el proceso de entrenamiento de un experimento específico. Permite al usuario conocer las pérdidas obtenidas en cada época durante la ejecución del experimento. Para obtener esta métrica, se deben especificar como parámetros de consulta tanto el nombre del experimento como el laboratorio en el que se ejecutó. Esta estadística ofrece una visión completa del entrenamiento, permitiendo al desarrollador utilizar estos datos para crear gráficos o tablas de resultados. La respuesta será un JSON que incluirá el laboratorio y ex-

perimento especificados, junto con todas sus medidas obtenidas durante el proceso de validación.

Por último, los desarrolladores tienen que tener la opción de **obtener los modelos entrenados**. Para habilitar esta capacidad, se sigue un procedimiento específico. En primer lugar, se identifica el *checkpoint* con el mejor rendimiento dentro del experimento indicado. Luego, se trasladan todos los archivos necesarios para cargar el modelo en el contenedor de Docker a la ruta */flogo/predict/model*. Estos archivos incluyen los pesos obtenidos durante el entrenamiento, almacenados en el *checkpoint* seleccionado, y la arquitectura utilizada, para determinar la distribución de los pesos entre las capas de la red neuronal. Una vez completados estos pasos, el directorio *model* se comprime en un archivo ZIP y se entrega al usuario a través del cuerpo de la respuesta.

Capítulo 6

Desarrollo

6.1. Herramienas

Durante el desarrollo de este TFT, se emplearon diversas herramientas tecnológicas que desempeñaron un papel fundamental en la consecución de los objetivos planteados. Se decidió hacer uso de estas porque brindaban una mayor facilidad en las tareas y se ajustaban mejor a las necesidades del proyecto. En esta sección, se detallarán de manera exhaustiva las herramientas utilizadas, justificando su elección y destacando su contribución al desarrollo y éxito de este proyecto.

6.1.1. Python

Durante todo el desarrollo de este proyecto, destacaron el uso de dos lenguajes de programación, siendo Python uno de ellos. Este es de un lenguaje de programación interpretado y de código abierto que posee orientación a objetos. Actualmente, suele ser de los más usado para enseñar a programar a aquellas personas que no posean conocimientos informáticos, gracias a su sintaxis clara y legible. Sin embargo, existen otros usos destacables más allá de los meros educativos. [11]

En las últimas décadas, Python ha surgido como una de las principales herramientas científicas para el procesamiento y visualización de grandes cantidades de datos, a pesar de no haber sido diseñado inicialmente para ello. Esta utilidad surgió a partir del desarrollo de otros paquetes, como NumPy, Pandas o Scikit-Learn, que dieron a los desarrolladores muchas herramientas con las que trabajar. Uno de esos campos donde el potencial de este lenguaje es explotado es el de DL, ámbito principal en el que nos enfocaremos en este TFT.

Teniendo en cuenta todo esto y el planteamiento inicial sobre el *framework*, se decidió que Python debía ser el lenguaje de programación en el que estaría desarrollado. Este aporta muchas facilidades en el trabajo con redes neuronales que no logran otros lenguajes, además de ser ampliamente usado y conocido por todos los programadores. [29]

6.1.2. Java

Java es un lenguaje de programación compilado que permite el desarrollo de aplicaciones de carácter general. Al igual que Python, este también está orientado a objetos y puede ser descargado de forma gratuita por cualquier persona. Actualmente, Java es uno de los lenguajes más usados en aplicaciones y servicios debido, principalmente, a su capacidad para reducir costos, acortar los plazos de desarrollo, impulsar la innovación y mejorar los servicios de aplicaciones. [8]

Durante estos últimos años, se ha podido contrastar muchas de estas ventajas que los propios desarrolladores aseguran tener frente a otras opciones en el mercado en diferentes proyectos. Por ello, Java suele ser una de las opciones principales cuando surge la necesidad de programar, como ha sido en este caso. Más concretamente, este lenguaje fue usado para la elaboración del servicio de laboratorio y su API asociada.

6.1.3. ItRules

Como se comentó en la Introducción, una de las bases principales de todo el proyecto era la elaboración de un DSL que permitiera un acceso mucho más sencillo y comprensible tanto al *framework* de diseño de redes neuronales como al que gestionaba su ciclo de vida. Para ello, dicho lenguaje específico de dominio tendría que ser capaz de generar el código necesario en Python para ejecutar los requerimientos especificados por el usuario. Esto fue posible gracias a las plantillas creadas con *ItRules*.

ItRules es un motor de plantillas basado en Java desarrollado en el Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (SIANI), capaz de generar código para el lenguaje de programación deseado a partir de una serie de reglas que sean definidas por el usuario. Un algoritmo interno se encargará de interpretar cada una de las reglas y serán las clases programadas en Java las que determinen cuáles de ellas deben ser lanzadas. [15]

Su corta curva de aprendizaje y su excelente rendimiento en comparación con otros motores de plantillas disponibles en el mercado fueron las razones principales por las que se eligió esta opción. Además, al ser un producto lanzado por una institución asociada a la universidad, existía la posibilidad de solicitar asesoramiento en caso de que fuera necesario. La principal utilidad de esta herramienta en el desarrollo de este proyecto fue crear las plantillas necesarias para que el compilador traduzca el código del DSL al código Python, donde se utilicen los *frameworks* desarrollados.

6.1.4. Docker

Durante el desarrollo del TFT surgió la necesidad de brindar a los usuarios aquellos modelos que habían sido entrenados con sus especificaciones. Esta acción debía ser sencilla para ellos, evitando que ninguno conociera el funcionamiento interno de nuestro *framework*;

ese modelo debía ser como una caja negra para ellos. Además, el modelo no se podía ver afectado por el hardware que poseyera cada uno, ya que debía funcionar en todos los sistemas de igual manera.

Docker [9] es una plataforma de software que permite crear, implementar y compartir aplicaciones de forma independiente a la infraestructura donde se esté ejecutando. Hace uso de tecnologías de contenedores para empaquetar y distribuir los programas que desee el desarrollador, compactando los mínimos recursos posibles para lograr ejecuciones consistentes en cualquier entorno.

Su capacidad de encapsular la ejecución de código en contenedores, permitiendo ejecuciones idénticas en sistemas con características distintas, y la amplia experiencia adquirida a lo largo de los años en el grado con esta herramienta, fueron esenciales para la selección de su uso por encima de otras posibilidades.

Docker se utilizó principalmente como herramienta para proporcionar los modelos entrenados dentro del servicio de laboratorio, listos para ser utilizados por el usuario. Estos recibirían todos los archivos necesarios para crear una imagen de *Docker*, en la cual el modelo estaría en ejecución, listo para recibir solicitudes.

6.1.5. PyTorch

Python, como se mencionó anteriormente, no proporciona todas las herramientas necesarias para trabajar de manera completa en el ámbito del DL. La potencia de este lenguaje radica en el desarrollo de un amplio conjunto de paquetes que enriquecen su funcionalidad. Uno de estos paquetes es PyTorch, que fue seleccionado para ser utilizado en la primera implementación de *Flogo*.

PyTorch es un *framework* desarrollado en Python que ofrece una amplia gama de herramientas rápidas y eficientes para el trabajo en el ámbito del aprendizaje profundo (DL). Entre sus características principales, destaca que permite a los usuarios crear y entrenar modelos de forma distribuida. Además, cuenta con una extensa comunidad de investigadores y desarrolladores que brindan conocimientos y apoyo a quienes lo necesiten. [24]

Una de las principales razones que motivaron la elección de este *framework* sobre otros fue la gran capacidad de personalización que ofrece, tanto en la creación de modelos como en su entrenamiento posterior. Gracias a su programación a un nivel más bajo que otras opciones del mercado, PyTorch proporciona una mayor flexibilidad para adaptarse a las preferencias o necesidades del usuario.

6.1.6. IntelliJ y PyCharm

Como ya ha sido mencionado, Python y Java han sido los dos lenguajes de programación elegidos para el desarrollo del TFT. Además, para cada uno de los lenguajes, se requiere un

entorno de desarrollo integrado (IDE) que permita escribir, editar, depurar y ejecutar el código. En este caso, los IDE seleccionados han sido *PyCharm* [19] e *IntelliJ* [18], principalmente debido a las experiencias satisfactorias tenidas con ambos en proyectos anteriores. Ambos son productos de la misma empresa, llamada *JetBrains*. Estos ofrecen un amplio conjunto de funcionalidades que reducen el tiempo necesario para la creación de una aplicación, como una amplia variedad de atajos de teclado (*shortcuts*) y un sistema de sugerencias.

6.1.7. Git

El control de versiones es una práctica esencial en el desarrollo de software colaborativo que permite realizar un seguimiento y gestionar los cambios en el código fuente a lo largo del tiempo. Esto se aplica para mantener un registro histórico de todas las modificaciones, facilitando la colaboración entre múltiples desarrolladores y asegurando la integridad y coherencia del proyecto. El control de versiones ayuda a evitar conflictos, permite revertir cambios problemáticos y coordina el trabajo en equipo. Para este proyecto, se optó por hacer uso de Git como herramienta de control de versiones.

Este es un sistema de control de versiones distribuido que permite a los desarrolladores realizar un seguimiento de los cambios en el código, colaborar de manera eficiente y gestionar diferentes versiones de un proyecto. A diferencia de los sistemas centralizados, Git proporciona a cada desarrollador una copia completa del repositorio, lo que facilita trabajar de manera autónoma y sin necesidad de una conexión constante a un servidor central. [20]

Gracias a esta herramienta, se registra cada modificación realizada al código, permitiendo revisar el historial de cambios. Cada versión del software se etiqueta con un identificador único, lo que facilita el seguimiento de qué cambios se hicieron, quién los hizo y cuándo se realizaron. Asimismo, Git no brinda la capacidad de crear ramas o *branches*, que son líneas paralelas de desarrollo. Estas permiten a los desarrolladores trabajar en nuevas funcionalidades o corregir errores de manera aislada del código principal. Posteriormente, estas ramas pueden fusionarse de nuevo en la rama principal, integrando los cambios de forma controlada.

Teniendo en cuenta el grado colaborativo de este proyecto, tener un buen control de versiones ha sido fundamental para lograr el avance eficiente y correcto de todo el código desarrollado.

6.1.8. GitHub

GitHub [12] es una plataforma de desarrollo colaborativo que utiliza el sistema de control de versiones Git. Esta permite a los desarrolladores trabajar de manera conjunta en proyectos, facilitando la gestión de cambios y la colaboración. Los repositorios de GitHub almacenan el código fuente de los proyectos, y cada cambio realizado en el código se registra con un historial detallado, lo que permite revertir cambios si es necesario y facilita la identificación de contribuciones específicas.

Teniendo en cuenta que este TFT a formado parte de un sistema mayor desarrollado entre otros trabajos finales de título, GitHub ha sido una herramienta de coordinación fundamental, permitiendo trabajar de forma independiente y unir las aportaciones de cada uno de forma sencilla. Asimismo, esta plataforma también permite compartir el proyecto de forma pública para que, todas aquellas personas que lo requieran, puedan hacer uso de él de forma gratuita.

6.2. Metodologías

Durante el desarrollo del proyecto, se han seguido una serie de metodologías y enfoques con el objetivo de asegurar la eficiencia, calidad y mantenibilidad del software. Estas han sido fundamentales para lograr un desarrollo ordenado y colaborativo, sobre todo teniendo en cuenta que, realmente, ha sido un proyecto conjunto desarrollado en 3 TFT distintos. Asimismo, se ha establecido como prioridad en todo momento intentar cumplir con los objetivos planteados al inicio del proyecto. En las siguientes secciones se presentan las metodologías usadas para abordar este proyecto.

6.2.1. Metodología ágil

Las metodologías ágiles [4] priorizan la flexibilidad, la colaboración y la adaptación continua a lo largo del ciclo de vida del proyecto. Se basan en un conjunto de principios y valores que promueven la entrega temprana y frecuente de software funcional, la respuesta rápida a los cambios del cliente y la cooperación estrecha entre los equipos de desarrollo y los clientes o usuarios finales. En contraste con los métodos tradicionales de desarrollo de software, que se centran en la planificación detallada y la ejecución secuencial, la metodología ágil aboga por ciclos cortos de desarrollo iterativo e incrementos de funcionalidad entregables en periodos de tiempo definidos, conocidos como *sprints*. [16]

Las metodologías ágiles se caracteriza por su naturaleza adaptativa, que se manifiesta en la capacidad de responder ágilmente a los cambios en los requisitos del proyecto, las preferencias del cliente o las condiciones del mercado. Esto se logra mediante la comunicación constante y la retroalimentación entre los miembros del equipo y los interesados, así como la realización de revisiones regulares del progreso del proyecto para identificar y abordar rápidamente cualquier desviación o ajuste necesario en la dirección del desarrollo. Además, los principios ágiles fomentan la autoorganización de los equipos y la toma de decisiones colaborativa, lo que permite una mayor agilidad y capacidad de respuesta ante situaciones imprevistas.

Esta metodología es especialmente útil en proyectos cuyas soluciones técnicas se desconocen, ya que los equipos de trabajo no son capaces de asegurar con rotundidad cuáles son las mejores soluciones a seguir. Esto acaba provocando cambios y la necesidad de revisar de forma continua con el cliente las nuevas implementaciones para ajustar el desarrollo a sus necesidades y prioridades. El hecho de tener *sprints* de corta duración permite adaptar posibles cambios y debatir posibles soluciones a los problemas encontrados de forma muy rápida y eficiente. En la Ilustración 6.1, se puede ver los principios básicos de la metodología.

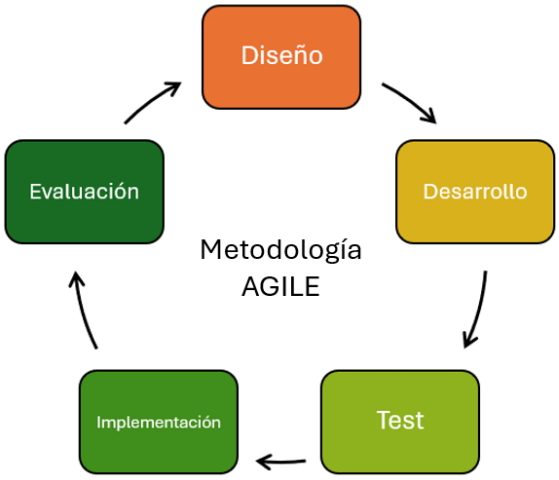

Ilustración 6.1: Principios básicos de la metodología ágil

Siendo más concreto, para este proyecto se ha seleccionado SCRUM, dentro de las metodologías ágiles disponibles. SCRUM es un marco de trabajo ágil utilizado en la gestión de proyectos y desarrollo de software. Se centra en la entrega incremental de productos a través de iteraciones llamadas *sprints*, que generalmente duran de dos a cuatro semanas. SCRUM promueve la colaboración entre equipos multifuncionales y autogestionados, la adaptabilidad ante cambios y la mejora continua mediante la revisión y retroalimentación constantes. Se ha escogido esta metodología por la experiencia previa trabajando con ella y por la elevada incertidumbre de los requisitos de la solución final. Esta incertidumbre obligaba a una adaptación constante del proceso de desarrollo, haciendo que SCRUM resultase una metodología idónea.

En resumen, la metodología ágil y adaptativa en el desarrollo de software es un enfoque centrado en la entrega temprana de valor, la colaboración multidisciplinaria y la capacidad de adaptación a los cambios. Al abrazar la incertidumbre inherente al desarrollo de software y fomentar la flexibilidad y la comunicación abierta, las metodologías ágiles permiten a los equipos responder de manera eficaz a los cambios del entorno y a las necesidades del cliente, manteniendo un enfoque constante en la calidad y la satisfacción del usuario.

6.2.2. Semantic versioning

Semantic Versioning (SemVer) [23] es un sistema de numeración de versiones que se utiliza principalmente en el desarrollo de software para especificar y comunicar cambios en el código de manera clara y consistente. Este sistema consta de tres números separados por puntos: *Major.Minor.Patch*. El número de versión principal (*Major*) se incrementa cuando se realizan cambios incompatibles con versiones anteriores. El número de versión secundario (*Minor*) se incrementa cuando se agregan nuevas funcionalidades de manera retrocompatible. El número de versión de parche (*Patch*) se incrementa cuando se realizan correcciones de

errores retrocompatibles. Además de estos números, SemVer permite el uso de etiquetas de versión para indicar cambios prelanzados o experimentales, así como metadatos adicionales para especificar información de compatibilidad y dependencias.

El uso de SemVer facilita la comprensión de la compatibilidad entre diferentes versiones de un software y ayuda a los desarrolladores y usuarios a tomar decisiones informadas sobre la actualización de sus aplicaciones. Al seguir un esquema de numeración coherente y significativo, SemVer promueve las buenas prácticas de desarrollo de software y contribuye a la estabilidad y la interoperabilidad de los sistemas informáticos al tiempo que reduce la probabilidad de conflictos y errores causados por actualizaciones inesperadas.

6.3. Cronología

Tras haber mencionado todas las herramientas utilizadas y las metodologías aplicadas, se procederá a describir detalladamente el procedimiento seguido en este proyecto. En esta descripción se podrán observar los pasos seguidos y cómo se aplicaron cada uno de los conceptos explicados. Como se mencionó anteriormente, para este proyecto se optó por utilizar la metodología ágil y adaptativa, estructurando las diferentes iteraciones en *sprints* de períodos de tiempo no superiores a dos semanas. En cada uno de estos *sprints*, se buscaba aportar valor de alguna manera y avanzar en la evolución del trabajo.

Al inicio de este proyecto, se enfrentó un desafío que se había manifestado a lo largo de los cuatro años de estudio: la automatización del diseño y entrenamiento de redes neuronales podía ser considerablemente mejorada en comparación con el estado actual. Para abordar esta cuestión, se dedicaron los primeros *sprints* a investigar el bajo uso de prácticas de ingeniería de software en el ámbito de la IA. Se llevó a cabo un análisis de las metodologías utilizadas por la comunidad científica en la creación de modelos, lo que confirmó la hipótesis inicial. Se evidenció la ausencia de un estándar claro en estos procesos, con cada programador adaptándose al *framework* elegido, a menudo pasando por alto principios básicos como la modularidad del código, lo que podría facilitar su comprensión y escalabilidad.

Las siguientes iteraciones se destinaron a intentar idear posibles soluciones que solventaran la problemática encontrada. Se revisaron muchos artículos en búsqueda de situaciones similares a la descrita que sirvieran de inspiración. Fue en este punto donde se encontraron desarrollos de software como los de GitHub Actions y Keras, comentados en la sección de Estado actual, que sirvieron como guía de posibles soluciones. Es importante destacar que este proceso no llegaba a finalizar completamente ya que, gracias al uso de metodologías ágiles y adaptativas, el diseño de la solución permaneció abierto, ajustándose continuamente a medida que surgían problemas y se identificaban nuevos requisitos necesarios.

Como resultado de este proceso, se llegó al acuerdo de desarrollar un *framework* que encapsularía el uso de los entornos de trabajo habituales de IA y permitiría la automatización de prácticas típicas que suelen llevar a errores. El usuario final se comunicaría con este a través de un DSL que reduciría la cantidad de conocimiento necesario para el programador para crear su propia red neuronal. Con el transcurso de varios *sprints*, surgió la necesidad de

dividir el *framework* en dos partes debido a su gran complejidad: uno para el diseño y otro para la gestión del ciclo de vida de las redes neuronales.

Una vez delineado el concepto general del producto final, se procedió a seleccionar las herramientas que lo materializarían. Para el *framework* operacional, se optó por Python, específicamente PyTorch, para respaldar la primera implementación. Esta elección se fundamentó en su capacidad para realizar cambios a un nivel más bajo nivel que otros entornos de trabajo con redes neuronales. Es importante destacar que se dedicó un *sprint* completo a esta decisión, dado su considerable impacto en el desarrollo del proyecto.

Las iteraciones posteriores se enfocaron en alcanzar una primera versión operativa del *framework* de entrenamiento de redes neuronales. Este proceso se dividió en distintos *sprints*, cada uno dirigido a desarrollar una funcionalidad específica que agregara valor en comparación con el estado anterior del proyecto. Esta etapa, que ocupó la mayor parte del tiempo, se centró en establecer una base sólida sobre la cual se pudieran agregar funcionalidades en futuras iteraciones. Hubo una colaboración estrecha con el desarrollo del otro *framework*, encargado del diseño de la arquitectura de redes neuronales, dada la relación cercana entre ambos. Se exploraron diversas implementaciones y enfoques, que se adaptaron con la ayuda de reuniones posteriores a cada *sprint*, para lograr una versión inicial funcional y escalable. La utilización de herramientas como Git fue fundamental para trabajar de manera autónoma y lograr una evolución eficiente y ágil.

Conforme avanzaban las iteraciones, surgían nuevas necesidades y requerimientos que no se habían contemplado inicialmente. Por ejemplo, los conceptos de laboratorio y experimentos no estaban incluidos desde el principio. Durante el desarrollo de *Flogo*, se identificó la necesidad de que los desarrolladores de DL realicen diversas pruebas con diferentes valores en los hiperparámetros o arquitecturas distintas para encontrar la más efectiva. Los experimentos y laboratorios fueron concebidos para facilitar este proceso de manera organizada y clara. Además, ofrecen la oportunidad de implementar algún tipo de automatización o heurística que simplifique esta tarea en futuras versiones. Trabajar con metodologías que consideran estos cambios como parte integral del desarrollo facilitó su incorporación al trabajo existente.

Otro de los requisitos detectados fue la necesidad de tener un sistema de almacenamiento de modelos. Durante los procesos de entrenamiento, muchas veces el modelo resultante en la última época no resulta ser el que mejor rendimiento obtiene, ya que existen muchas circunstancias que pueden reducir el acierto de una red neuronal en sus predicciones, como el *overfitting* o una tasa de aprendizaje muy alta. Por ello, se decidió dedicar una serie de iteraciones a la implementación de un sistema de guardado de que permitiera almacenar todos aquellos modelos que mejoraran el rendimiento de sus predecesores. De esta forma, el usuario tendría la opción de usar el conjunto de mejores pesos y no solo aquellos que fueron obtenidos en la última época.

Otro proceso importante fue el desarrollo de las plantillas utilizadas por el generador de código, permitiendo que, a partir del modelo especificado con el DSL, se ejecutara lo solicitado por el usuario. Para lograr esto, la coordinación entre los procesos de creación del DSL y el *framework* fue esencial a lo largo de las iteraciones de esta etapa. Fue necesario acordar las interfaces necesarias para gestionar el ciclo de vida de la red neuronal.

Una vez alcanzada una versión del *framework* que cumplía la mayoría de los objetivos establecidos, se dio inicio al desarrollo del servicio de laboratorios. Este servicio permitiría a los usuarios trabajar de manera más fluida con los archivos Python generados por el proceso de compilación y gestionar los diferentes modelos. Se comenzó delineando las operaciones necesarias que este servicio web debía soportar, así como las tecnologías a emplear, considerando las necesidades básicas de los usuarios. En los siguientes *sprints*, se implementaron gradualmente estas operaciones utilizando Java, y se refinaron las versiones basándose en el *feedback* recibido en las reuniones.

Una vez completadas las implementaciones de los dos principales softwares de este TFT, las últimas iteraciones se dedicaron a realizar pruebas de integración de todo el sistema, a pesar de que al final de cada *sprint* se llevaba a cabo una verificación para asegurar que los nuevos cambios no afectaran el funcionamiento del sistema. Posteriormente, se inició la fase de evaluación de los resultados obtenidos en todo el proyecto, considerando las conclusiones alcanzadas y los productos desarrollados. En el Cuadro 6.1, se muestra una distribución más concisa de los *sprints* realizados y sus principales actividades asociadas.

Cuadro 6.1: Distribición del trabajo realizado en sus respectivos *sprints*

Conjunto de sprints	**Trabajo realizado**
Conjunto de sprints 1	Análisis del problema
Conjunto de sprints 2	Diseño de posibles soluciones
Conjunto de sprints 3	Selección de herramientas a usar
Conjunto de sprints 4	Diseño de los conceptos de laboratorio y experimentos
Conjunto de sprints 5	Implementación de un sistema de guardado de *checkpoints*
Conjunto de sprints 6	Desarrollo de plantillas con *ItRules*
Conjunto de sprints 7	Desarrollo de la primera versión funcional del servicio de laboratorio
Conjunto de sprints 8	Pruebas de integración

Capítulo 7

Conclusiones y trabajo futuro

El problema abordado ha llevado a la obtención de diferentes conclusiones con respecto al proyecto. Para poder exponerlas de forma manera completa, se mencionarán cuáles han sido las aportaciones que más destacan del proyecto, además de comentar qué aprendizajes, tanto personales como académicos, hemos obtenido durante la elaboración. Asimismo, veremos con qué grado se han logrado cumplir los objetivos iniciales y qué propuestas se tienen para continuar mejorando *Flogo* en futuras iteraciones.

7.1. Resultados

Con este documento, se da por finalizado una primera iteración de este proyecto. Este se inició con una idea vaga de cómo sería el resultado final, pero con convicciones claras sobre lo que se quería lograr. A partir de ahí, comenzó un proceso de investigación, donde, apoyados por los conocimientos prácticos y teóricos adquiridos en los últimos cuatro años, se logró adentrarse en el mundo del DL y alcanzar una solución válida de acuerdo con los requisitos iniciales.

El trabajo sobre el problema abordado se ha traducido en una serie de productos finales que pueden ser usados por todas aquellas personas interesadas. El primero es el eje sobre el que ha girado todo este documento, que es un *framework* para la gestión del ciclo de vida de una red neuronal. Este se encarga principalmente de todos los procedimientos que engloba una red neuronal desde que es diseñada hasta que está lista para ser usada en producción, como su entrenamiento, su testeo, la prueba de diferentes hiperparámetros, etc.

Asimismo, este *framework* conforma una parte de un sistema mayor, llamado *Flogo*, desarrollado de forma conjunta con otros dos TFT. Este tiene por objetivo facilitar a los desarrolladores el trabajo con redes neuronales en muchos ámbitos, permitiéndoles, principalmente, abstraerse del conocimientos de otros entornos de DL y la automatización de muchos procesos típicos y repetitivos.

Todo esto ha sido publicado a través de la plataforma de *GitHub*, donde cualquier persona

puede acceder a estos programas y hacer uso de ellos si es necesario. En dicho repositorio, se podrán encontrar manuales de uso que indican cómo se debe hacer uso tanto de *Flogo* como de cada uno de los *frameworks* que lo componen de forma independiente.

Igualmente, la servicio de laboratorios, destinado a facilitar el trabajo con *Flogo*, también es un resultado final. Esta da la posibilidad a los usuarios de desentenderse de la gestión de todas sus arquitecturas y laboratorios, permitiéndoles tener su propio repositorio y transmitir la responsabilidad de almacenar los resultados y los modelos entrenados en el servidor donde se ejecute la el servicio.

Más allá de los productos realizados en este proyecto, se ha logrado que este sea usado en una situación real. En el Instituto Europeo de Investigación Energética (EIFER) se desarrolló con *Flogo* una red neuronal para modelar la demanda eléctrica de un microgrid consumidos en una zona determinada. Esto resulta un paso importante, ya que se logra demostrar que el sistema es capaz de funcionar más allá del ámbito académico.

Para realizar una valoración general de los resultados del proyecto, es primordial analizar los objetivos planteados al inicio y evaluar el grado de cumplimiento alcanzado. El objetivo general y más significativo era elaborar un *framework* accesible para el entrenamiento de redes neuronales, que permitiese la personalización de dicho proceso para diversos contextos y aplicaciones. Este se puede considerar validado, puesto que se ha logrado desarrollar el *framework* para la gestión del ciclo de vida de una red neuronal, incluyendo el proceso de aprendizaje de los pesos. Este *framework* es accesible para quien lo desee y permite la personalización completa de todo el procedimiento. Igualmente, es importante analizar el grado de cumplimiento de los objetivos secundarios.

Primeramente, se debe destacar que se ha logrado alcanzar el diseño extensible y modular que se planteó al inicio, permitiendo tener un *framework* altamente escalable y adaptable a nuevas tecnologías. Asimismo, se ha implementado la mayor parte de las técnicas y algoritmos de entrenamiento y optimización de redes neuronales, ofreciendo a los usuarios las mejores opciones en el mercado actualmente. Este *framework* es también adaptable y compatible con una amplia gama de arquitecturas, por lo que se da por alcanzada esta meta también.

Otro aspecto importante desde el inicio radicaba en la capacidad del *framework* para adaptarse e integrarse con herramientas ya existentes. Durante el desarrollo, se centraron muchos de nuestros esfuerzos en lograr que el *framework* sea completamente abierto a la integración con la tecnología deseada por el usuario final, por lo que se puede asegurar haber completado este propósito.

En cuanto a las visualizaciones y análisis de modelos, se ha conseguido que el usuario sea capaz de acceder a los resultados obtenidos por sus modelos en los procesos de entrenamiento, validación y prueba. Sin embargo, un punto no completado es la utilización de gráficas para mostrar estos resultados de forma más visual. Esto puede ser un punto de mejora o expansión importante en el futuro.

Sin embargo, existen algunos objetivos que no se han podido cubrir en el resultado final. Por ejemplo, no se ha logrado completar el proceso de automatización de la selección de los hiperparámetros de una red neuronal, ya que el diseño de una heurística resultó ser un

proyecto de investigación mucho más extenso. En contraposición, se implementó el concepto de experimento, que permite variar el valor de los hiperparámetros y facilita las comparaciones entre las configuraciones seleccionadas.

En el Cuadro 7.1, se puede ver un pequeño resumen con el grado de cumplimiento de cada uno de los objetivos iniciales que acaban de ser explicados.

Cuadro 7.1: Revisión de los objetivos iniciales del proyecto

Referencia	**Objetivo**	**Grado de cumplimiento**
Objetivo 1	Diseño modular y extensible	Cumplido
Objetivo 2	Interfaz sencilla e intuitiva	Cumplido
Objetivo 3	Automatización y optimización de hiperparámetros	No cumplido
Objetivo 4	*Framework* actualizado	Cumplido
Objetivo 5	Compatibilidad entre redes	Cumplido
Objetivo 6	Evaluación y análisis de modelos	Parcialmente cumplido
Objetivo 7	Compatibilidad e integración con herramientas existentes	Cumplido

7.2. Contribuciones

Estos productos finales desarrollados contribuyen de muchas maneras a todas aquellas personas que hagan uso de estos. Algunas de las principales ventajas que aporta este sistema serán mencionadas a continuación.

Flogo ha resultado ser un sistema que facilita y automatiza muchos de los pasos seguidos en el diseño y entrenamiento de redes neuronales. Antes de la elaboración de estos *frameworks*, existían muchas metodologías extendidas en toda la comunidad que hacían de estas técnicas, procesos tediosos y sólo aptos para aquellos con conocimientos suficientes para entender cada mínimo fallo que pudiera surgir. Con este trabajo, se ha logrado reducir la complejidad y el tiempo necesario para crear modelos que puedan ser usados por los desarrolladores. Asimismo, durante su elaboración, han surgido nuevas inquietudes que pueden acabar dando lugar a interesantes y valiosos trabajos de investigación en el futuro.

Se ha logrado reducir la curva de aprendizaje que tienen aquellas personas que se quieran adentrar dentro del ámbito del DL a través de la simplificación y estandarización de muchos de los procesos que se ejecutan durante el entrenamiento de una red neuronal. La inexistencia actual de un estándar dentro de esta rama perjudica a la comunicación de los resultados y el intercambio de código entre los investigadores, puesto que cada uno sigue su propia metodología. Por ello, sería fundamental alcanzar criterios de diseños comunes para favorecer la evolución de la IA.

Igualmente, el desarrollo del lenguaje específico de dominio de *Flogo* ha permitido que el código necesario para diseñar y entrenar una red neuronal sea mucho más entendible por todas las personas. Esto ha favorecido que estos ficheros sean muchos más explicativos sobre

los procedimientos que se están ejecutando, a pesar de no tener grandes conocimientos sobre DL. De esta forma, se logra que la posible comunicación de posibles resultados o arquitecturas sea mucho más sencilla.

El sistema desarrollado también permite a los programadores explorar, de manera mucho más eficiente y ordenada, diferentes tipos de arquitecturas o conjuntos de hiperparámetros en busca de aquellos que den el mejor rendimiento. Anteriormente, este procedimiento era bastante tedioso y caótico, conduciendo a errores leves que no permitían obtener los mejores resultados posibles.

En definitiva, el resultado final resulta gratificante, no solo por el hecho de haber cumplido con la mayor parte de las expectativas iniciales, si no por el hecho de haber logrado unir dos de los campos más importantes de nuestro grado en un único trabajo, como son la Ingeniería del Software y la IA. Se espera que esta actividad académica sirva como un punto de partida para futuras investigaciones que busquen soluciones a los desafíos que enfrentamos.

7.3. Aprendizajes

En cuanto a los aprendizajes educativos, se pueden destacar muchas experiencias que han contribuido significativamente a nuestra formación como persona y profesional. Una de ellas ha sido la oportunidad de llevar a cabo un proyecto de una envergadura superior a los trabajos habituales que se desarrollan durante el transcurso del grado. Resulta interesante y enriquecedor tener la posibilidad desarrollar un trabajo a largo plazo, siguiendo los pasos habituales que se toman en el mundo laboral para este tipo de casos. Tuvimos que aprender a abordar un proyecto de esta dimensiones, empezando por plantear y delimitar adecuadamente cuál iba a ser el problema a tratar, con el objetivo de establecer que debía cumplir el producto final.

Una vez definido el desafío, diseñamos como lo enfocaríamos y cuáles serían nuestro primeros pasos para intentar alcanzar la solución. Al ser un proyecto tan extenso y complejo, muchas propuestas fracasaron, pero fueron estos errores los que nos enseñaron el camino a seguir. Estas experiencias no han sido comunes en nuestras trabajos académicos previas, lo que hace que el TFT sea una oportunidad única para aplicar y vivir un aprendizaje práctico de manera significativa.

También es importante destacar el aspecto educativo de trabajar en un proyecto cuyo objetivo es desarrollar un sistema práctico y de utilidad para la comunidad. Establecer desde un inicio que estamos creando un marco de trabajo accesible para todos, representa un cambio significativo en nuestra perspectiva sobre el problema. Es importante tener esta primera experiencia antes de entrar en el mercado laboral puesto que, en la mayoría de las ocasiones, las empresas buscan profesionales que sean capaces de transferir sus conocimientos a la práctica.

Otro aprendizaje que se ha obtenido durante este proyecto ha sido la capacidad para trabajar en equipo. Como ya se ha descrito en repetidas ocasiones, el resultado final de este TFT conforma una pieza más de un sistema, llamado *Flogo*, que se completan con el trabajo

de otros dos compañeros. Llevar a cabo una elaboración coordinada y conjunta de cada uno de los componentes ha supuesto un reto para nosotros. Por ello, para asegurar el éxito del proyecto hemos tenido aprender a trabajar de forma conjunta, adaptándonos cada uno a la forma de trabajar del resto y aprendiendo tomar decisiones como grupo.

También se debe mencionar la curiosidad que este proyecto a despertado en nosotros por la investigación dentro del ámbito de la informática y la IA. En muchas ocasiones, nos hemos enfrentado a situaciones nuevas donde los métodos de solución no estaban claros. Esto nos ha servido como introducción al trabajo que realiza un investigador, aunque sea de forma mucho más reducida. Además, este proyecto no ha abierto las puertas a nuevas ideas que resultan realmente interesante para desarrollar en futuros trabajos, para continuar adentrándonos en el mundo de la investigación.

Finalizando con los aprendizajes obtenidos durante este proceso de elaboración del TFT, debemos mencionar todos los conocimientos técnicos que ha sido adquiridos durante su elaboración. Por ejemplo, hemos tenido nuestra primera experiencia con un DSL, que resulta ser un instrumento fundamental y altamente útil dentro de la informática. A través de este, hemos visto por primera vez herramientas de generación de código, esenciales para la automatización de muchos procesos.

7.4. Trabajo futuro

El producto final resultante de este TFT es plenamente funcional y puede ser usado, en su estado actual, por desarrolladores para administrar el ciclo de vida de sus modelos de DL. Sin embargo, siempre existen aspectos que pueden ser mejorados o nuevas funcionalidades que se pueden añadir. A continuación, se comentarán algunas de las opciones por donde se puede continuar mejorando el proyecto.

El primer punto a trabajar en un futuro cercano sería ampliar aún más el *framework*, añadiendo nuevos métodos de entrenamiento, nuevos optimizadores, nuevas funciones de pérdida, etc. El DL es una rama en constante evolución, y estar actualizado es esencial para no quedarse atrás con respecto a otros entornos. Asimismo, actualmente no se permite entrenar todo tipo de arquitecturas, por lo que sería bastante importante continuar el desarrollo para ofrecer el mayor número de opciones posibles a los usuarios.

Como se mencionó en los objetivos iniciales, una de las ideas antes de comenzar con el trabajo era la de añadir la opción de que un usuario fuera capaz de ver gráficas que muestren de forma visual cómo se comportan sus modelos con los datos con los que han sido entrenados. Esta opción resulta muy interesante y añadiría gran valor a la aplicación final, puesto que reduciría aún más la curva de aprendizaje de las personas que quieran comenzar a trabajar con redes neuronales. Otro paso más sería elaborar un cuadro de mando con todos los resultados obtenidos para un modelo. De esta forma, se podría proporcionar a los clientes un informe completo de cómo se ha desarrollado el entrenamiento.

Otra opción para mejorar sería la creación de un entorno en línea de desarrollo que permitiera a los usuarios interactuar con todas las partes que conforman el sistema de *Flogo*

de manera mucho más sencilla. De esta forma, se lograría que los desarrolladores no tuvieran que trabajar con archivos o con una API, sino que todos estos procedimientos serían aspectos automáticos del servicio.

Un aspecto importante dentro de la IA que no se ha mencionado en este documento es lograr la máxima eficiencia posible de los recursos del ordenador durante el entrenamiento de modelos. Estos procesos suelen ser altamente costosos debido al gran número de operaciones que se requieren para obtener una red neuronal funcional. Por tanto, un punto de trabajo futuro sería buscar esta eficiencia a través, por ejemplo, de la implementación de la paralelización de experimentos. De esta forma, se lograría reducir considerablemente los tiempos de entrenamiento y maximizar el uso de los recursos disponibles en el ordenador.

La difusión del trabajo realizado también es un aspecto fundamental dentro de cualquier proyecto. La elaboración de un artículo científico o *paper*, donde se exponga el trabajo realizado y las conclusiones obtenidas, y la creación de cursos donde se enseñe a las personas interesadas cómo hacer uso de *Flogo*, ayudarían a que el sistema sea más conocido y, por tanto, aumentaría su valor.

Sin embargo, la opción de mayor mejora y que daría mayor valor al producto final está relacionada con la automatización de la asignación de hiperparámetros. Como se mencionó al inicio, es posible elaborar una heurística que permitiera conocer cuáles son los mejores hiperparámetros para entrenar una arquitectura determinada. Con ella, se podría evitar el proceso de prueba de diferentes optimizadores, funciones de pérdidas, funciones de activación, etc., en busca de aquella que dé los mejores resultados, ya que sería el *framework* el que indicaría al usuario cuáles debe usar. Este sería un proyecto de investigación ambicioso cuyos resultados serían muy importantes para la comunidad científica.

Bibliografía

[1] (2024). Github actions documentation. https://docs.github.com/en/actions. Accedido: 17-10-2024.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283. Accedido: 17-10-2024.

[3] Aggarwal, C. C. (2018). *Neural Networks and Deep Learning: A Textbook*. Springer.

[4] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. Accedido: 24-10-2024.

[5] Botev, Z. I., Kroese, D. P., Rubinstein, R. Y., and L'Ecuyer, P. (2013). The cross-entropy method for optimization. *Handbook of statistics*.

[6] Chai, T. and Draxler, R. (2014). Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature. *Geoscientific Model Development*.

[7] Chollet, F. et al. (2023). Keras: The python deep learning library. https://keras.io. Accedido: 17-10-2024.

[8] Corporation, O. (2024). Java documentation. https://www.oracle.com/java/. Accedido: 17-10-2024.

[9] Docker (2024). What is a container? https://www.docker.com/resources/what-container/. Accedido: 17-10-2024.

[10] Evans, C., döt Net, I., and Ben-Kiki, O. (2009). Yaml ain't markup language (yaml) official website. Accedido: 2024-06-06.

[11] Foundation, P. S. (2024). Python documentation. https://www.python.org/about/apps/. Accedido: 17-10-2024.

[12] GitHub (2024). About github. https://docs.github.com/en/github. Accedido: 17-10-2024.

[13] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT press.

[14] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

[15] Hernández Cabrera, J. J., Évora Gómez, J., Roncal Andrés, O., and Caballero Ramirez, M. (2016). Itrules. https://bitbucket.org/siani/itrules-java/wiki/Home. Accedido: 17-10-2024.

[16] Highsmith, J. (2004). *Agile Project Management: Creating Innovative Products*. Addison-Wesley Professional, Boston, MA.

[17] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R*. Springer, New York.

[18] JetBrains (2024a). Intellij idea. https://www.jetbrains.com/es-es/idea/. Accedido: 17-10-2024.

[19] JetBrains (2024b). Pycharm. https://www.jetbrains.com/es-es/pycharm/. Accedido: 17-10-2024.

[20] Loeliger, J. and McCullough, M. (2012). *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc.

[21] Naveed, H., Arora, C., Khalajzadeh, H., Grundy, J., and Haggag, O. (2023). Model driven engineering for machine learning components: A systematic literature review.

[22] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

[23] Preston-Werner, T. (2024). Semantic versioning.

[24] PyTorch (2024). Pytorch features. https://pytorch.org/features/. Accedido: 17-10-2024.

[25] Pérez, J. D. R. (2024a). Flogo framework. https://github.com/NeuralFlogo/framework. Accedido: 10-06-2024.

[26] Pérez, J. D. R. (2024b). Laboratory service. https://github.com/NeuralFlogo/Laboratory-service. Accedido: 10-06-2024.

[27] Ruder, S. (2016). An overview of gradient descent optimization algorithms.

[28] Team, D. (2024). Data version control: Git for data & models. Accedido: 17-10-2024.

[29] VanderPlas, J. (2016). *Python Data Science Handbook. Essential Tools For Working With Data*. O'Reilly Media.

[30] Zeiler, M. D. (2012). Adadelta: An adaptive learning rate method.

Glosario

CPU Componente principal de una computadora responsable de interpretar y ejecutar instrucciones de programas, realizando cálculos y gestionando tareas esenciales para el funcionamiento del sistema. 17

dataset conjunto de datos organizados y estructurados que se utilizan para realizar análisis, investigaciones o entrenar modelos en diversas áreas. 19

Deep Learning Conjunto de técnicas de inteligencia artificial orientadas a crear programas de computador que puedan aprender de la experiencia. 2

Dockerfile Script de texto que contiene una serie de instrucciones para construir una imagen de Docker de manera automatizada. 43

DSL Lenguaje de programación con un nivel superior de abstracción optimizado para una clase específica de problemas. 2

endpoints description . 37

framework Conjunto estandarizado de criterios, prácticos, conceptos y herramientas para abordar una problemática particular. 2

GPU Componente de hardware especializado en acelerar el procesamiento y renderizado de gráficos y realizar cálculos paralelos, muy usado en el entrenamiento de redes neuronales. 17

IDE Aplicación de software que ayuda a los programadores a desarrollar código de software de manera eficiente. 50

Keras Biblioteca de redes neuronales de código abierto escrita en Python, especialmente diseñada para posibilitar la experimentación con redes de Aprendizaje profundo. 8

MD5 Protocolo criptográfico que se usa para autenticar mensajes y verificar el contenido y las firmas digitales basado en una función de HASH. 42

Model Driven Engineering Paradigma de ingeniería de software, el cual se centra en la creación y explotación de modelos de dominio. 9

MPS Tecnología de NVIDIA que implementa la API CUDA, una plataforma NVIDIA que admite computación GPU de uso general. 17

outliers Observaciones que numéricamente son muy distante del resto de los datos. Se consideran valores atípicos. 67

shortcuts Tecla o conjunto de teclas que efectúa acciones definidas previamente con el objetivo de ahorrar tiempo. 50

TensorFlow Plataforma de extremo a extremo de código abierto desarrollada por Google, enfocada en el aprendizaje automático. 2

Theano Biblioteca de *Python* y un compilador de optimización para manipular y evaluar expresiones matemáticas, especialmente las que incluyen valores matriciales. 9

YAML Formato de serialización de datos legible por humanos inspirado en lenguajes como *XML*, *C*, *Python*. 8

Apéndice A

Funciones de pérdidas

A continuación, aprovecharemos este anexo para explicar algunas de las funciones de pérdidas más importantes dentro del mundo de redes neuronales. Todas ellas fueron han sido implementadas en nuestro *framework*, por lo que se encuentran disponibles para todos sus usuarios. Como podrán ir viendo, se harán uso de fórmulas para facilitar su explicación y, a menos que se indique lo contrario, se usarán las siguientes variables:

$$y_i \rightarrow \text{Valor observado o esperado en el momento } i$$
$$\hat{y}_i \rightarrow \text{Valor predicho por el modelo en el momento } i$$

A.1. Error cuadrádito medio

El error cuadrático medio, también llamado como *Mean Squarred Error* (MSE) [17], es una de las funciones de pérdidas más usadas en los modelos de *Deep Learning*, especialmente efectivas en regresiones. Esta se puede definir como el promedio de los cuadrados de las diferencias entre el valor que haya sido predicho por el modelo y el esperado. En otras palabras, calcula, de media cuanto se equivoca el modelo con cada una de sus predicciones. Su fórmula es la siguiente:

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{A.1}$$

El cuadrado en esta fórmula es usado por diversas razones. Una de las principales fue para lograr eliminar signos negativos y evitar cancelaciones con errores de diferentes signos, proporcionando una medida más adecuada de la magnitud total de los errores. Asimismo, su implementación provoca que los errores grandes sean mucho más penalizados que los pequeños, que suelen ser más problemáticos y costosos. Por último, también facilita el uso de técnicas de optimización, como el descenso de gradiente, para ajustar los parámetros del modelo.

A.2. Error absoluto medio

Error absoluto medio o, más comúnmente conocido como *Mean absolute error* (MAE) [6] es un método que copia la idea base de la explicada anteriormente. Por ello, esta también mide las diferencias promedio entra las salidas indicadas por el dataset y las predichas por la red neuronal. La diferencia erradica en el uso del valor absoluto de las diferencias en lugar de un elevado al cuadrado. Por tanto, la ecuación queda de la siguiente manera:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{A.2}$$

Con el uso del valor absoluto, a parte de lograr mantener el hecho de evitar problemas con resultados con signos distintos que se anulen, logramos dos características que permiten a este método diferenciarse. Por un lado, el resultado obtenido es más comprensible para las personas, puesto que los valores se encuentran en la misma escala (no están elevados al cuadrado). Por otro lado, es más robusto a *outliers* que otros métodos, por lo que sus resultados se ven menos influenciados por casos atípicos.

A.3. Huber

La función de pérdidas de *Huber* [14] se puede definir como una combinación de los dos métodos mencionados anteriormente. Esta combina las ventajas de la función de pérdida cuadrática en la proximidad del punto de predicción correcto y las ventajas de la función de pérdida absoluta en regiones más lejanas. Su ecuación queda definida como:

$$L(y,\hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{para } |y-\hat{y}| \leq \delta \\ \delta(|y-\hat{y}| - \frac{1}{2}\delta) & \text{para } |y-\hat{y}| > \delta \end{cases}$$

donde tenemos que definir δ como un parámetro que define el valor diferénciala partir del que se lleva a cabo la transición entre ambas funciones. Cuando la diferencia entre el valor verdadero y el valor predicho es pequeña (menor o igual a δ), la función se comporta como la función de pérdida cuadrática, mientras que cuando la diferencia es grande (mayor que δ), se comporta como la función de pérdida absoluta.

Huber es, como las funciones que lo compenen, especialmente útil en modelos que vayan a resolver un problema de regresión. Además, se caracteriza por ser especialmente útil cuando se tienen datos que pueden contener *outliers* o valores atípicos que pueden afectar negativamente a la eficacia del modelo.

A.4. Divergencia de Kullback-Leibler

La divergencia de *Kullback-Leibler* (KL) [5] es una función de pérdidas que calcula de forma numérica la distancia o diferencia existente entre dos distribuciones de probabilidad. Para ello, esta función mide la cantidad promedio de información adicional necesaria para codificar las salidas como una distribución real en lugar de una predicha por un modelo. Su fórmula es la siguiente:

$$KL(P\|Q) = \sum_{x} P(x) \log\left(\frac{Q(x)}{P(x)}\right) \tag{A.3}$$

Siendo $P(x)$ la distribución de probabilidad real y $G(x)$ la predicha. Destacar como características esenciales que se trata de una función de pérdidas asimétrica, ya que, como podemos ver en la fórmula, $\mathrm{KL}(P\|Q)$ es distinto a $\mathrm{KL}(Q\|P)$ Igualmente, se trata de otro método que también penaliza fuertemente las predicciones incorrectas en comparación con las predicciones cercanas a la distribución real.

A.5. Entropía cruzada

La entropía cruzada, o *cross-entropy* [5] , es una función de pérdidas, basada en el método anterior, usada principalmente en tareas de clasificación. Para su cálculo, se mide la diferencia existente entre la distribución de probabilidad real, es decir, la de los valores reales, y la distribución predicha por el modelo. Su fórmula es la siguiente:

$$CE = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{A.4}$$

Como es lógico al ser una función de pérdidas que se tratará de minimizar, para valores bajos de esta fórmula se dice que las distribuciones son bastante parecidas, por lo que las salidas esperadas y las del modelo tiene pocas diferencias. Su principal ventaja es que penaliza fuertemente las predicciones que están muy lejos de las etiquetas verdaderas. Por este motivo, este método está bueno en la clasificación, ya que en estos problemas no solo importa qué clase se predice, sino también como de seguro está el modelo de la misma.

Apéndice B

Optimizadores

En esta sección extra, veremos la mayor parte de los optimizadores que se encuentran implementados actualmente dentro de Flogo. Durante su definición, se harán uso de varias fórmulas para describirlas correctamente, por lo que, antes de comenzar su explicación, debemos definir una serie de variables:

$$w_t \rightarrow \text{Parámetros del modelo en el momento } k$$
$$\nabla J \rightarrow \text{Gradiente de la función objetivo}$$
$$x_k \rightarrow \text{Salidas del modelo en el momento } k$$
$$y_k \rightarrow \text{Resultados esperados en el momento } k$$
$$\rho \rightarrow \text{Tasa de aprendizaje}$$

B.1. SGD

El optimizador *Stochastic Gradient Descent*(SGD) [13], es uno de los algoritmos más usados que hay dentro de la rama de *Machine Learning* para el ajuste de los parámetros de una red neuronal. Su funcionamiento se basa en el descenso del gradiente, que es uno de los métodos básicos para la obtención del punto mínimo dentro de una función. En este, se iteran cada uno de los parámetros en la dirección opuesta a la marcada por el gradiente, ya que este es un vector que indica la dirección del cambio más pronunciado en una función. [27] De esta manera, se alcanzan los valores óptimos del modelo. Su función es la siguiente:

$$w_{t+1} = w_t - \rho \cdot \nabla J(w; x_k, y_k)$$

El concepto de estocástico proviene de que no se calcula el gradiente sobre todos los datos usados en esa iteración, sino que se escoge una muestra de forma aleatoria. Esto hace al algoritmo mucho más rápido, pero puede introducir una gran variación en el modelo y causar fluctuaciones mientras se actualiza el parámetro.

B.2. SGD Nesterov

El *SGD Nesterov* [13], también conocido como *NAG* se trata de una variante del *SGD* explicado anteriormente, donde se añade el concepto de *momentum* o momento para mejorar la convergencia, es decir, para encontrar de forma más rápida la solución óptima. En cuanto al *momentum*, este lo podemos definir como una técnica donde se añade una fracción del vector de movimiento de la iteración anterior a la actual. En otras palabras, se recuerda la actualización de los pesos en cada iteración para que la siguiente sea una combinación lineal del gradiente y la actualización anterior.

El impulso puede ser muy bueno, pero si es demasiado alto, el algoritmo puede pasar por alto los mínimos locales. Por ello, en este algoritmo se añade otra modificación. En este algoritmo, se intenta anticipar al movimiento llevado a cabo por el *momentum*. Para ello, *NAG* primero da un gran salto en la dirección del gradiente calculado en la iteración anterior $(w - \gamma \cdot V(k-1))$ y luego continua el proceso. Su fórmula queda así:

$$w_{t+1} = w_t - (\gamma \cdot V(k-1) + \rho \cdot \nabla J(w - \gamma \cdot V(k-1); x_i, y_i))$$

Siendo γ un hiperparámetro conocido como momentum o inercia y $V(k-1)$ el vector de movimiento en el k anterior. Con este optimizador, somos capaces de crear un equivalente a la inercia en el movimiento, ya que se tienen en cuenta los pasos seguidos en iteraciones anteriores. Este método logra suavizar las oscilaciones en direcciones de baja curvatura y permite un movimiento más rápido en direcciones de alta curvatura.

B.3. AdaGrad

AdaGrad [13] se planteó como un nuevo punto de mejora del *SGD*. En este caso, la novedad erradicaría en el valor de la tasa de aprendizaje. En los métodos anteriores, este hiperparámetro se había mantenido estable, es decir, no sufría variaciones durante las ejecuciones. Con Adagrad se planteó que cada peso tuviera su propio tasa, cuyo valor vendría determinado por el acumulado de los gradientes en iteraciones anteriores:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2 + E}} \cdot \nabla J(w; x_k, y_k)$$

Siendo g_t el gradiente obtenido en el instante t y η el valor inicial dado. [30] De esta forma, se logra que los valores de la tasa de aprendizaje vayan adaptándose en función de las necesidades del modelo en ese momento. En épocas iniciales, la tasa tomaría valores altos para dar grandes saltos. Conforme fueran pasando las iteraciones y nos fuéramos acercando al mínimo, este hiperparámetro se iría haciendo más pequeño.

B.4. RMSProp

El igual que en los casos anteriores, *RMSProp* [13] se presentó como una mejora a uno de los algoritmos ya vistos. El principal inconveniente que tenía *AdaGrad* es que el valor de la tasa de aprendizaje se volvía bastante pequeño en iteraciones avanzadas. Por ello, en este método se cambió la forma en la que se acumulaban los gradientes anteriores, pasando a hacerse un promedio exponencialmente decreciente, donde os valores más recientes tienen más influencia en el resultado final que los valores anteriores.

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2 + E}} \cdot \nabla J(w; x_k, y_k)$$

Otra ventaja que aporta este método es que ya no se tienen que almacenar los n gradientes anteriores, ya que este se calcula de forma recursiva. Es decir, usando el valor de la iteración anterior.

B.5. Adadelta

El algoritmo de *Adadelta* [30] comparte muchas similitudes con el *RMSProp*. Este también fue creado como una extensión de *Adagrad*, diseñada para reducir su decaimiento excesivo de la tasa de aprendizaje. La diferencia erradica en que, mientras *RMSProp* usa una media móvil exponencial de los cuadrados de los gradientes anteriores para la actualización, *Adadelta* limita la ventana de acumulación a un tamaño fijo, incorporando una media móvil de las actualizaciones de los parámetros. Su fórmula es la siguiente:

$$w_{t+1} = w_t - \frac{\sqrt{\sum_{\tau=1}^{t-1} \nabla J_\tau^2 + E}}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2 + E}} \cdot \nabla J(w; x_k, y_k)$$

Su principal ventaja erradica en que *Adadelta*, como vemos en la fórmula, elimina la necesidad de elegir manualmente una tasa de aprendizaje global, ya que las unidades de actualización son autoajustadas basadas en las estadísticas de los gradientes recientes.

B.6. Adam

El algoritmo de *ADAM* [13] se define como una combinación de dos optimizadores ya vistos, *RMSProp* y *SGD* con momentum. Por un lado, usa el mismo método que *RMSProp* para calcular el valor de la tasa de aprendizaje en cada iteración. Por otro lado, copia el método de impulso o inercia que posee el *SGD* con *momentum* para buscar de forma más rápida los puntos mínimos. Con esta combinación, *ADAM* se convierte en un optimizador con cambios de movimientos menos bruscos.