Trabajo de Fin de Grado

Framework para la Implementación de Arquitecturas de Redes Neuronales

TITULACIÓN: Grado en Ciencia e Ingeniería de Datos

AUTOR: Juan Carlos Santana Santana

TUTORIZADO POR
José Juan Hernández Cabrera

Junio 2024

Agradecimientos

A mi familia, en concreto a mi madre y mi padre, por todo el apoyo que me han dado siempre.

Resumen

Este proyecto de fin de grado se enmarca dentro de un sistema denominado *Flogo* que permite simplificar la implementación de redes neuronales, aislando a los desarrolladores de los detalles técnicos de las librerías de aprendizaje profundo. Su objetivo es permitir que los desarrolladores se centren en la lógica y estructura de las redes a desarrollar, reduciendo de esta forma la complejidad y promoviendo la eficiencia y colaboración entre equipos con diferentes conocimientos. Este sistema fomenta la estandarización y la reutilización del código, lo que se traduce en desarrolllos más rápidos y productos de mayor calidad. En concreto, este trabajo expone el desarrollo del framework que usa el sistema para la implementación de las arquitecturas de redes neuronales.

Abstract

This final degree project is part of a system called *Flogo* that simplifies the implementation of neural networks, isolating developers from the technical details of deep learning libraries. Its goal is to allow developers to focus on the logic and structure of the networks to be developed, thus reducing complexity and promoting efficiency and collaboration between teams with different expertise. This system encourages standardization and code reuse, resulting in faster development and higher quality products. Specifically, this paper presents the development of the framework used by the system for the implementation of neural network architectures.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

> Depender en el sentido de la abstracción. Los módulos de alto nivel no deben depender de detalles de bajo nivel.
>
> Robert C. Martin

A finales del siglo pasado, las redes neuronales se convirtieron en un tema prevalente en el ámbito de la Inteligencia Artificial (IA) y del *Machine Learning* (ML), debido a la invención de diversos métodos de aprendizaje eficientes y arquitecturas de red innovadoras. Aunque las redes neuronales se utilizaron con éxito en numerosas aplicaciones, el interés en la investigación sobre este tema disminuyó posteriormente.

Sin embargo, la introducción del *Deep Learning* (DL) [6] consiguió un renacimiento de la investigación en este ámbito por el hecho de que las redes profundas, al ser entrenadas adecuadamente, lograron demostrar un éxito significativo en una variedad de desafíos.

En cuanto al ámbito de trabajo, como se observa en la figura 1.1, el DL se considera un subconjunto del ML y de la IA. De este modo, el ML puede verse como una función de la IA que imita el procesamiento de datos del cerebro humano diferenciándose del DL en que este último incrementa su eficiencia a medida que aumenta el volumen de datos. Esto es debido a que las redes neuronales utilizan múltiples capas para representar las abstracciones de los datos y construir modelos computacionales. Es por ello que, aunque el DL requiere un tiempo prolongado para entrenar un modelo debido al gran número de parámetros, el tiempo de ejecución durante las pruebas es corto en comparación con otros algoritmos de IA.

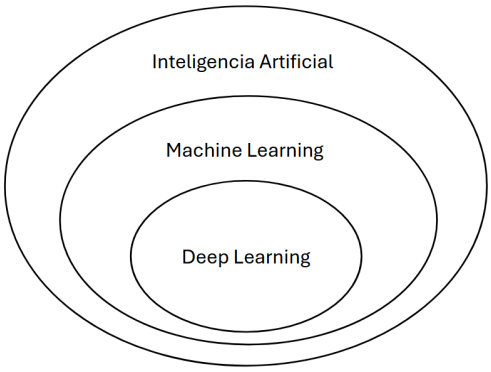

Figura 1.1: Posición del DL en comparación con el ML y la IA.

Por esta razón, en el panorama actual de la IA, la implementación efectiva de redes neuronales constituye un componente fundamental para numerosas aplicaciones tanto para labores de investigación y desarrollo como para entornos de producción y comercialización. El auge en este ámbito ha sido catalizado por la convergencia de varios factores, entre los que se incluyen el aumento en la disponibilidad de datos, el desarrollo de arquitecturas más eficientes y las tecnologías de bajo coste donde, según la conocida *Ley de Moore*, aproximadamente cada dos años se duplica el número de transistores en un microchip, lo que permite aumentar el rendimiento y reducir los costos de producción de dispositivos electrónicos, haciendo que las tecnologías avanzadas sean cada vez más accesibles y económicas. Este renovado interés en las redes neuronales ha generado un impulso sin precedentes en la investigación, así como un creciente interés por parte de la industria en la aplicación de estas tecnologías para abordar problemas complejos en áreas como la visión por computador, el procesamiento del lenguaje natural y el reconocimiento de voz, entre otros [11].

Sin embargo, la complejidad inherente a la configuración y optimización de estas redes ha creado barreras significativas para muchos desarrolladores. Entre los problemas más destacables, se encuentra la necesidad de elección de la librería a usar de entre las múltiples existentes para la producción de modelos de DL, así como las pronunciadas curvas de aprendizaje presentes en las mismas, requiriendo de una comprensión profunda de los detalles y procesos específicos de cada una. Adicionalmente, como en cualquier otro producto software, las redes neuronales pueden sufrir problemas como la dificultad en el escalado, la obsolescencia producida por un escaso mantenimiento o la incapacidad de adaptación a la introducción de nuevas características y actualizaciones en el ámbito.

En respuesta a este desafío, la solución propuesta se lleva a cabo a través de un sistema denominado *Flogo* [17] que, como se puede ver en la figura 1.2 se fundamenta en la utilización de un *Domain-Specific Language* (DSL) diseñado para describir tanto el diseño como el ciclo de vida de una red neuronal. Este DSL actúa como una herramienta de alto nivel que permite a los desarrolladores especificar de manera precisa la arquitectura y el comportamiento deseado de la red neuronal. A partir de esta descripción en el DSL, el sistema se apoya en unas plantillas que permiten la generación de código específico para ambos módulos. Por un lado, el *framework* estructural encargado de crear el esqueleto de la red neuronal, definiendo su estructura y conexiones internas. Y, por otro, el *framework* operacional dedicado al proceso

de entrenamiento que se encarga de implementar el algoritmo de aprendizaje y ajuste de los parámetros de la red. Como resultado de este proceso, se produce una red neuronal entrenada que se encuentra preparada para su uso en producción para realizar inferencias. De esta manera, el sistema facilita el desarrollo y entrenamiento de redes neuronales, automatizando tareas complejas y proporcionando una metodología estructurada para su implementación eficiente.

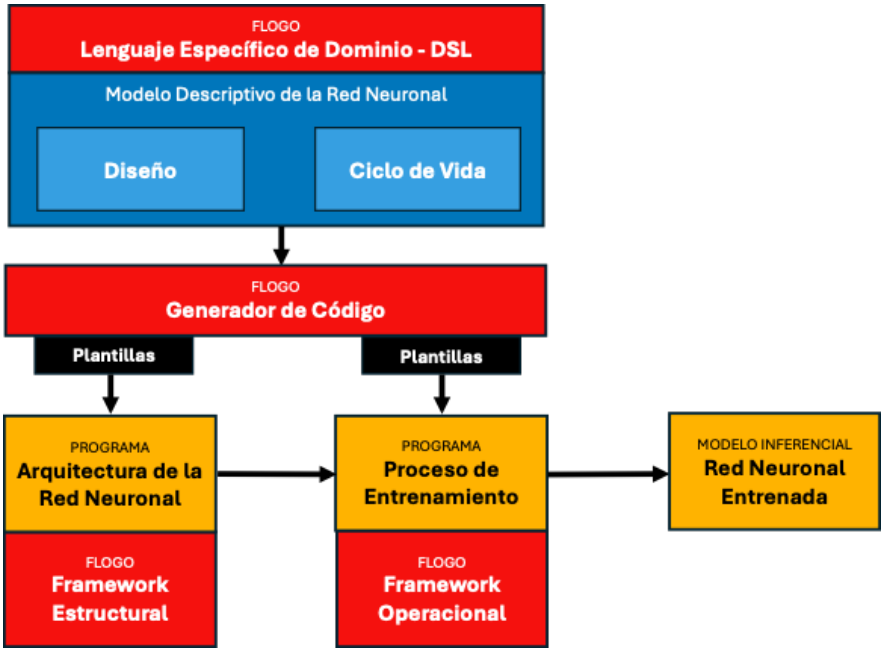

Figura 1.2: Estructura de componentes de *Flogo*.

El propósito fundamental del sistema radica en simplificar el proceso de implementación y puesta en producción de modelos de redes neuronales al proporcionar una capa de abstracción sobre las diversas librerías existentes en el ámbito de la IA. En lugar de requerir un profundo conocimiento técnico de cada una de estas librerías, los desarrolladores usan *Flogo* para enfocarse en la estructura y lógica de las redes neuronales, aislando la carga asociada con los detalles de implementación específicos de cada librería. De esta forma, este trabajo permite promover la accesibilidad al desarrollo de aplicaciones de DL, al eliminar las barreras de entrada para aquellos desarrolladores que carecen de experiencia en el uso de librerías específicas y, al proporcionar un marco unificado y coherente para la implementación de redes neuronales, fomentando la estandarización y la reutilización de código, lo que puede resultar en un desarrollo más rápido y una mayor calidad en los productos finales.

Como se ha comentado anteriormente, el sistema está compuesto por diferentes módulos diferenciados, siendo la parte que concierne a este proyecto el *framework* estructural en el que se apoya el DSL para la definición de las diferentes arquitecturas de redes neuronales. Este *framework* es, por tanto, homomorfo con el DSL y funciona como capa de aislamiento entre el lenguaje definido y las diferentes librerías existentes para la implementación de redes neuronales. De esta forma, proporciona una interfaz unificada para la definición de dichas arquitecturas, lo que facilita enormemente el proceso de diseño y desarrollo. Al abstraer la

complejidad subyacente de las diversas librerías y proporcionar una capa de aislamiento, permite a los desarrolladores enfocarse en la lógica de alto nivel de sus modelos sin preocuparse por los detalles de implementación específicos de cada una. Además, al ser homomorfo con el DSL, garantiza coherencia y consistencia en la sintaxis y la semántica, lo que mejora la legibilidad y la mantenibilidad del código. Es, por tanto, un componente crucial del sistema, puesto que es la base que permite al DSL generar las arquitecturas con el fin de proveérselas al *framework* operacional para su posterior refinamiento.

En los capítulos siguientes, se expondrá una visión general del estado actual y los objetivos iniciales para posteriormente proceder a explorar en detalle el diseño del *framework*, la integración del mismo con el resto de componentes del sistema, una serie de ejemplos de uso y su proceso de desarrollo, finalizando con las conclusiones a las que se ha llegado con el proyecto y una descripción del trabajo futuro planeado.

Antes de continuar, destacar que en el campo de la inteligencia artificial e ingeniería del software es común usar términos en inglés debido a su amplia adopción y falta de traducción en muchas ocasiones. Por ello, se informa a los lectores que se utilizarán varios anglicismos o acrónimos de los mismos que serán explicados en detalle en el glosario.

Capítulo 2

Contexto

En este capítulo se examina el panorama actual en el ámbito del DL, exponiendo la problemática existente en el uso de las librerías con mayor relevancia para la producción de modelos en dicho campo. Posteriormente, se presentan los objetivos iniciales del proyecto de fin de grado y las competencias específicas cubiertas con el mismo.

Los expertos en el ámbito de la IA hacen uso de las librerías predominantes ya existentes en el mercado, como son *PyTorch* [16] o *TensorFlow* [1], para la producción de modelos de DL debido a la amplia adopción, extensiva documentación y soporte que poseen. Estas librerías no solo facilitan la producción de modelos de DL, sino que también ofrecen herramientas avanzadas para la investigación y el desarrollo, lo que las convierte en elecciones populares entre los profesionales del sector. Aunque existen otras librerías relevantes y también populares, *PyTorch* y *TensorFlow* destacan por su versatilidad y capacidad para abordar una amplia variedad de problemas en este ámbito. Ambas han desempeñado roles significativos en el impulso y la popularización del aprendizaje automático, cada una con sus propias fortalezas y características distintivas. Sin embargo, a pesar de su indudable potencia, su uso efectivo no está exento de desafíos.

2.1. Estado actual

Entre las limitaciones que presentan estas librerías, se destaca la imposición de estructuras y paradigmas de diseño rígidos, lo que puede obstaculizar la flexibilidad y creatividad de los expertos en la implementación de soluciones innovadoras. Esta dependencia excesiva de las librerías preexistentes no solo limita la capacidad de personalización de los modelos, sino que también puede conducir a la estandarización y homogeneización de las soluciones, reduciendo así la diversidad y originalidad en la investigación y desarrollo en este campo.

Por otro lado, la pronunciada curva de aprendizaje de estas herramientas puede resultar en un aumento de la complejidad y el tiempo necesario para implementar y depurar redes neuronales, constituyendo una barrera de entrada para nuevos usuarios que se enfrentan a la necesidad de familiarizarse con las peculiaridades de cada una. Las figuras 2.1 y 2.2 muestran

ejemplos de arquitecturas implementadas con dichas librerías, en los cuales se puede apreciar fácilmente la dificultad inherente en su comprensión.

```
class MyNet(nn.Module):
  def __init__(self):
    super(MyNet, self).__init__()
    self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
    self.relu1 = nn.ReLU()
    self.pool1 = nn.MaxPool2d(2, stride=2)
    self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
    self.relu2 = nn.ReLU()
    self.pool2 = nn.MaxPool2d(2, stride=2)
    self.fc1 = nn.Linear(512, 10)
    self.softmax = nn.Softmax(dim=1)

  def forward(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)
    x = x.view(-1, 512)
    x = self.fc1(x)
    x = self.softmax(x)
    return x
```

Figura 2.1: Red neuronal implementada en *Pytorch*.

```
class MyNet(tf.keras.Model):
  def __init__(self):
    super(MyNet, self).__init__()
    self.conv1 = tf.keras.layers.Conv2D(64, kernel_size=3, strides=1, padding='same')
    self.relu1 = tf.keras.layers.ReLU()
    self.pool1 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=2)
    self.conv2 = tf.keras.layers.Conv2D(128, kernel_size=3, strides=1, padding='same')
    self.relu2 = tf.keras.layers.ReLU()
    self.pool2 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=2)
    self.flatten = tf.keras.layers.Flatten()
    self.fc1 = tf.keras.layers.Dense(10)
    self.softmax = tf.keras.layers.Softmax()

  def call(self, x):
    x = self.conv1(x)
    x = self.relu1(x)
    x = self.pool1(x)
    x = self.conv2(x)
    x = self.relu2(x)
    x = self.pool2(x)
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.softmax(x)
    return x
```

Figura 2.2: Red neuronal implementada en *Tensorflow*.

Por otro lado, las redes neuronales, como cualquier otro producto software, sufre de los problemas que estos acometen. De esta forma, la rápida evolución de estas herramientas a menudo implica la introducción de nuevas características y actualizaciones imponiendo la necesidad de adaptarse a dichas actualizaciones constantes para mantenerse al día con las mejores prácticas y optimizaciones. Esto puede resultar en una inversión considerable de tiempo y recursos que probablemente supongan retrasos en el desarrollo de proyectos y la imposibilidad de adaptación a nuevas demandas del mercado.

Este problema se ve reflejado en la figura 2.3 donde se representan las diferentes fases que tienen lugar en cualquier desarrollo software. En primer lugar se encuentra la fase de desarrollo, la cual incluye el diseño, la codificación, las pruebas y la implementación del software. El coste de esta fase es el más alto, ya que implica la creación del producto desde cero. Seguidamente, se encuentra la fase de producción, la cual incluye la distribución del software a los usuarios, siendo el coste de esta fase menor, ya que solo implica la corrección de los posibles errores surgidos. Por último, se produce la fase de mantenimiento en donde se suele dar el software como implementado y se han de realizar modificaciones simples, llegando al punto de realizar un mantenimiento mínimo. Sin embargo, a partir de este punto, pueden surgir problemas debido a la falta de mantenimiento y a la acumulación de errores no resueltos, conocidos como deudas técnicas. Esto resulta en un aumento de costes del producto con el tiempo, lo que puede llevar al abandono del proyecto y la necesidad de comenzar nuevamente desde el principio.

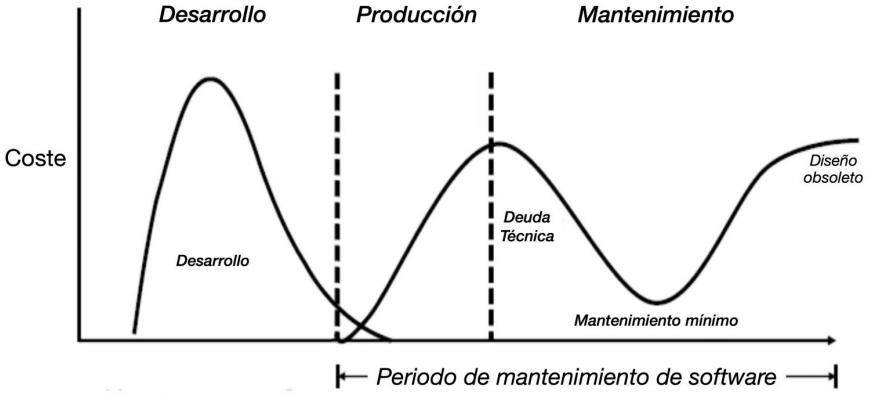

Figura 2.3: Coste de desarrollo de software según la fase en la que se encuentra.

2.2. Objetivos iniciales

El objetivo de este proyecto es proporcionar una capa intermedia que permita a los desarrolladores aislarse de los detalles técnicos de las plataformas subyacentes, permitiéndoles centrarse en la lógica y la arquitectura de sus modelos de DL. Esta aproximación no solo busca resolver los problemas asociados con las librerías existentes, sino que también tiene el potencial de facilitar el acceso al campo de la IA a nuevos usuarios.

En el contexto de este trabajo de fin de grado, es relevante mencionar que ya se ha realizado investigación previa relacionada con esta idea, ya que otros investigadores también han identificado el problema abordado, elaborando un estudio donde destacaron las principales tendencias y futuras líneas de trabajo relacionadas con el uso del enfoque de *Model Driven Engineering* (MDE) [18] dentro del ámbito [15]. No obstante, este proyecto se distingue por presentar una solución novedosa y original, que aporta un enfoque innovador para resolver la problemática en cuestión.

La tabla 2.1 presenta los objetivos propuestos para este trabajo de fin de grado, cada uno identificado con un código único. De este modo, el objetivo *O1* consiste en proporcionar a los desarrolladores un mayor control sobre sus proyectos, permitiéndoles diseñar redes neuronales según sus necesidades específicas sin estar limitados por las soluciones preexistentes. Este control fomenta la capacidad de adaptarse rápidamente a las necesidades cambiantes del mercado y facilita la innovación.

El objetivo *O2* destaca la importancia de una plataforma flexible que permita modificar y ajustar las estructuras de red de acuerdo con las últimas tendencias y tecnologías emergentes, sin la necesidad de esperar actualizaciones de terceros, lo cual es esencial para mantener la relevancia y eficiencia en un campo tan dinámico como la IA.

El objetivo *O3* busca asegurar la capacidad de mantener y actualizar las redes neuronales de manera independiente, sin depender de la disponibilidad o soporte de librerías comerciales específicas. Esto disminuye el riesgo asociado a las fluctuaciones del mercado y permite una mayor estabilidad y continuidad en el desarrollo de proyectos.

El objetivo *O4* se centra en la creación de una interfaz fácil de usar y altamente personalizable, que permita diseñar redes neuronales que se ajusten perfectamente a los requisitos específicos.

Cuadro 2.1: Objetivos propuestos para el trabajo de fin de grado.

Código	Objetivo
O1	Mayor control y autonomía
O2	Flexibilidad y adaptabilidad
O3	Independencia tecnológica
O4	Facilidad de uso y personalización

2.3. Competencias específicas

El presente trabajo de fin de grado ha permitido abordar una serie de competencias específicas de la carrera. A continuación, se enumerarán y analizarán detalladamente cada una de estas competencias, aclarando la relación de la misma con el trabajo.

- **EF4** “Conocimiento de la estructura, organización, funcionamiento e interconexión de los sistemas informáticos, los fundamentos de la programación, y su aplicación para

la resolución de problemas propios de la ingeniería." [4]. Este proyecto ha facilitado la comprensión de cómo interconectar diferentes componentes del sistema para crear una solución integrada y eficiente, demostrando la capacidad de utilizar estos conocimientos para resolver problemas complejos.

- **EC3** "Conocimiento y aplicación de los principios fundamentales y técnicas básicas de los sistemas inteligentes y su aplicación práctica." [4]. El proyecto ha servido como aplicación de los principios fundamentales y técnicas básicas de los sistemas inteligentes, integrando conceptos teóricos con aplicaciones prácticas. De esta forma, se ha facilitado la comprensión y utilización de algoritmos y modelos de ML, demostrando cómo estos principios pueden ser aplicados para diseñar y optimizar sistemas inteligentes capaces de resolver problemas complejos
- **EC4** "Conocimiento y aplicación de los principios, metodologías y ciclos de vida de la ingeniería del software." [4]. El diseño del *framework* creado está basado en los principios y metodologías de la ingeniería del software. Este proyecto ha implicado seguir los distintos ciclos de vida de desarrollo del software, desde la planificación y análisis de requisitos, pasando por el diseño y la implementación, hasta las fases de prueba y mantenimiento. La experiencia adquirida en este trabajo ha sido fundamental para comprender y aplicar prácticas de ingeniería del software que aseguran la calidad, eficiencia y sostenibilidad del código desarrollado, demostrando la importancia de estas metodologías en la resolución de problemas complejos y en la creación de soluciones tecnológicas robustas.
- **ED1** "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" [4]. A lo largo de este proyecto, se ha explorado en profundidad no solo los fundamentos teóricos de las redes neuronales, sino también las complejidades prácticas asociadas con su implementación efectiva en entornos informáticos. El análisis, diseño y construcción de este *framework* ha permitido una comprensión completa de los paradigmas y técnicas esenciales para el desarrollo de sistemas inteligentes. Al abordar los desafíos específicos relacionados con la integración de diferentes librerías de IA, se ha logrado un enfoque integral y robusto que permite a los desarrolladores aprovechar al máximo el potencial de las redes neuronales en una variedad de aplicaciones. Este trabajo ha demostrado cómo las técnicas de sistemas inteligentes pueden ser aplicadas de manera efectiva en el ámbito de la informática, brindando soluciones innovadoras y escalables para problemas complejos en diversas áreas de aplicación.
- **ED4** "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" [4]. Por un lado, se han identificado y analizado los problemas actuales enfrentados por los desarrolladores, como la falta de control sobre el diseño de las redes y la dependencia de actualizaciones de terceros. En respuesta, se ha ofrecido una solución que proporciona mayor autonomía y flexibilidad, facilitando el diseño y desarrollo de redes neuronales adaptadas a las necesidades específicas y tendencias emergentes. Por último, se ha realizado una documentación

detallada de la solución con el fin de facilitar el entendimiento a nuevos usuarios.

Capítulo 3

Construcción del framework

El propósito de este capítulo es abordar el diseño del *framework*, tanto sus principios de diseño como los constructos creados relacionando ambos con la implementación realizada, centrándose en el constructo de arquitectura como estructura principal para el entrenamiento y generación de los subsiguientes modelos.

3.1. Principios de diseño

Durante el desarrollo, la adherencia a principios de diseño sólidos ha sido crucial para garantizar la mantenibilidad, escalabilidad y flexibilidad del software. El *framework* presentado en este trabajo ha sido cuidadosamente diseñado siguiendo los principios *SOLID* [14] que constituyen una guía para la estructuración de software robusto. Estos principios no solo facilitan la extensión y adaptación del *framework* a nuevas necesidades sin comprometer su estabilidad, sino que también aseguran una arquitectura cohesiva y modular. La implementación de estos principios contribuye significativamente a la capacidad para soportar el rápido avance en el campo de las redes neuronales.

En primer lugar, el *Single Responsability Principle* (SRP) establece que una clase debe tener una única responsabilidad, es decir, solo una razón para cambiar. Este principio asegura que las clases sean cohesivas y que los cambios en una funcionalidad específica no afecten a otras partes del sistema, facilitando su mantenimiento y reduciendo la complejidad de las modificaciones.

El *Open/Close Principle* (OCP) indica que una clase debe estar abierta para la extensión pero cerrada para la modificación. Esto significa que el comportamiento de una clase se puede extender sin alterar su código fuente, típicamente mediante la utilización de la herencia o la composición, permitiendo añadir nuevas funcionalidades sin modificar el código existente.

El *Liskov Substitution Principle* (LSP) sugiere que los objetos de una clase derivada deben poder reemplazar a los objetos de una clase base sin afectar la funcionalidad del sistema. Este principio asegura que las jerarquías de clases se comporten de manera predecible y coherente.

El *Interface Segregation Principle* (ISP) dicta que los clientes no deberían verse forzados a depender de interfaces que no usan. Este principio sugiere la creación de interfaces específicas y pequeñas, en lugar de interfaces grandes y generales. De este modo, se evita que las clases dependan de métodos que no necesitan, promoviendo una mayor modularidad y reduciendo las dependencias necesarias.

Por último, el *Dependency Inversion Principle* (DIP) establece que los módulos de alto nivel no deben depender de módulos de bajo nivel, sino que ambos deben depender de abstracciones. Asimismo, las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones. Este principio enfatiza la importancia de depender de interfaces o clases abstractas en lugar de clases concretas, lo que aumenta la flexibilidad y reduce el acoplamiento entre componentes.

Otro principio clave en el diseño del *framework* ha sido la Ley de Demeter[13] también conocida como el principio de menor conocimiento, que sugiere que un objeto debe interactuar solo con objetos estrechamente relacionados. Su implementación implica diseñar las interacciones entre los diferentes componentes de la arquitectura de tal manera que cada objeto solo se comunique con sus colaboradores directos. Por ejemplo, la clase que define la arquitectura de la red neuronal no debe conocer los detalles internos de las capas individuales ni interactuar directamente con los parámetros de estas capas. En su lugar, debería utilizar métodos de alto nivel proporcionados por las capas para configurar y obtener los resultados necesarios.

El acoplamiento a este principio permite mantener las interacciones entre objetos limitadas a las estrictamente necesarias, reduciendo la dependencia entre diferentes partes del código, lo que facilita la modificación y ampliación del *framework* sin afectar a otras partes. Además, permite un mayor encapsulamiento, ya que cada componente puede ocultar sus detalles internos y exponer solo lo necesario, mejorando la modularidad y la claridad del código.

Los principios de diseño implementados en el desarrollo del *framework* han permitido una abstracción efectiva de las librerías subyacentes, como son *Pytorch* y *TensorFlow*. Gracias a estos principios, el *framework* es capaz de interactuar con ambas librerías de manera transparente, ofreciendo una interfaz unificada y consistente. Esto no solo facilita el desarrollo y mantenimiento del código, sino que también permite a los desarrolladores cambiar o actualizar las librerías subyacentes sin afectar significativamente la funcionalidad del *framework*, promoviendo así la flexibilidad y escalabilidad del sistema.

3.2. Diseño

El constructo de arquitectura actúa como eje central al cual se han de adherir los componentes y funciona como estructura principal para el entrenamiento y la generación de un modelo posterior. Es crucial distinguir entre los constructos de modelo y arquitectura en este contexto. Por un lado, arquitectura se refiere a un conjunto de capas organizadas en etapas de procesamiento que posibilitan el aprendizaje, mientras que un modelo consiste en una

arquitectura junto con un conjunto específico de pesos asociados. Esta distinción es fundamental para comprender el proceso de desarrollo y entrenamiento de redes neuronales dentro del *framework* propuesto.

La arquitectura de una red neuronal profunda típica contiene varias capas ocultas, incluidas las capas de entrada y salida. La figura 3.1 muestra una estructura general de una red neuronal profunda, la cual contiene mínimo dos capas ocultas, en comparación con una red superficial que contiene una única capa oculta.

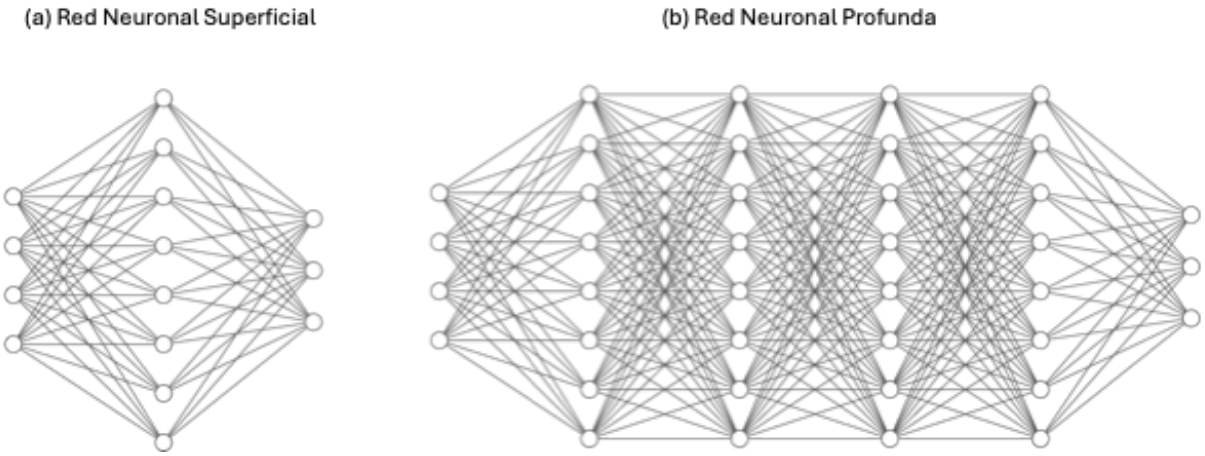

Figura 3.1: Arquitectura general de **a** una red neuronal superficial y **b** una red neuronal profunda con múltiples capas ocultas.

En este contexto, resulta fundamental comprender la estructura y organización interna de las diferentes arquitecturas de redes neuronales existentes. A continuación, se detallan las principales variantes de arquitecturas de redes neuronales utilizadas en la actualidad, teniendo en cuenta que, dada la amplia gama de redes neuronales existentes, se ha enfocado el desarrollo en aquellas consideradas más relevantes y prevalentes.

En primer lugar, las *Artificial Neural Networks* (ANN) [2] son reconocidas como la base de las redes neuronales profundas, y a lo largo del tiempo se han empleado diversas denominaciones para referirse a ellas, inicialmente conocidas como perceptrón multicapa y actualmente referidas como *Dense Network* o *Fully-connected*. Una ANN típicamente consiste en una red que comprende una capa de entrada para recibir datos de entrada, una capa de salida para realizar una decisión o predicción sobre la señal de entrada, y una o varias capas ocultas entre ellas, consideradas como el motor computacional de la red.

Siguiendo el orden de importancia, las *Convolutional Neural Networks* (CNN) [12] son un tipo de redes que pueden aprender características muy abstractas de los objetos, especialmente de datos espaciales, y puede identificarlos más eficientemente mediante la organización de diversas capas en una jerarquía. Un modelo CNN consiste en un conjunto finito de capas de procesamiento que pueden aprender varias características de datos de entrada como son imágenes, con múltiples niveles de abstracción. Las capas iniciales aprenden y extraen

características de bajo nivel, mientras que las capas más profundas aprenden y extraen características de alto nivel.

Por último, las *Recurrent Neural Networks* (RNN) [9] son un tipo de red neuronal diseñada para modelar datos secuenciales, es decir, datos en los que existen dependencias o relaciones entre los elementos posteriores y anteriores. A diferencia de las redes neuronales profundas tradicionales, que asumen independencia entre las entradas y salidas respecto a las anteriores, en las RNN la salida depende de elementos previos dentro de la secuencia. Estas redes poseen una "memoria" inherente, ya que utilizan información de entradas anteriores para influir en la entrada y salida actuales, lo que se logra mediante una capa oculta que retiene información a lo largo del tiempo.

Cada una de estas arquitecturas ofrece enfoques distintos para abordar problemas específicos en el ámbito del DL, desde la capacidad de las redes neuronales artificiales para aprender representaciones no lineales de datos, hasta la eficiencia de las convolucionales en el procesamiento de datos de alta dimensionalidad y el modelado de secuencias temporales mediante las recurrentes. Estas arquitecturas han servido como la base fundamental para la abstracción de componentes realizada en el *framework*, permitiendo la creación de una infraestructura flexible y adaptable para la creación de arquitecturas.

Con ello, como se puede observar en la figura 3.2, una arquitectura está compuesta por secciones, las cuales a su vez son una agregación de bloques, los cuales contienen un conjunto de capas específico.

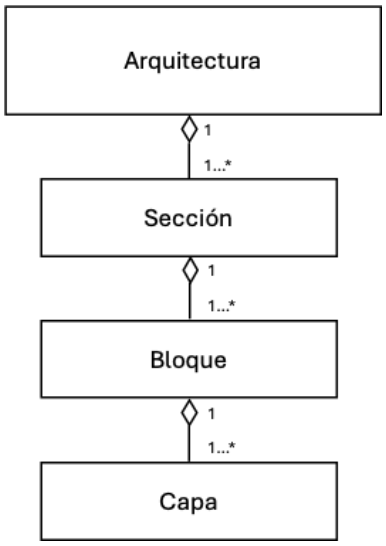

Figura 3.2: Diagrama de clases del framework en UML.

3.3. Capa

El componente básico de cualquier arquitectura son las capas. Estas se refieren a un conjunto de neuronas o unidades que se encuentran interconectadas entre sí y procesan la

información de entrada, lo que da lugar a la generación de una salida. Cada capa puede tener un número variable de neuronas, y está conectada a la capa anterior y posterior a través de conexiones ponderadas. Asimismo, tal y como se observa en la figura 3.3, una neurona es una unidad básica de procesamiento que trata de simular el comportamiento de una neurona biológica en el cerebro humano, recibiendo estímulos de entrada y generando una salida, la cual se obtiene al multiplicar dicha entrada por unos pesos aprendibles, agregándolos, y procesando una respuesta según el procedimiento que tiene definido, transmitiendo dicha respuesta al exterior.

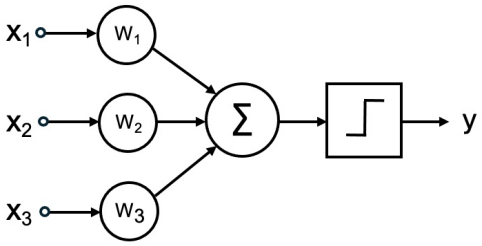

Figura 3.3: Comportamiento de una neurona.

Existe una amplia diversidad de capas que son usadas en las diferentes arquitecturas de redes neuronales que han sido estudiadas. Este conjunto de capas proporciona la flexibilidad y la versatilidad necesarias para construir y personalizar un amplio espectro de arquitecturas de redes neuronales existentes hasta la fecha. Hasta el momento se han implementado las capas que se observan en la figura 3.4.

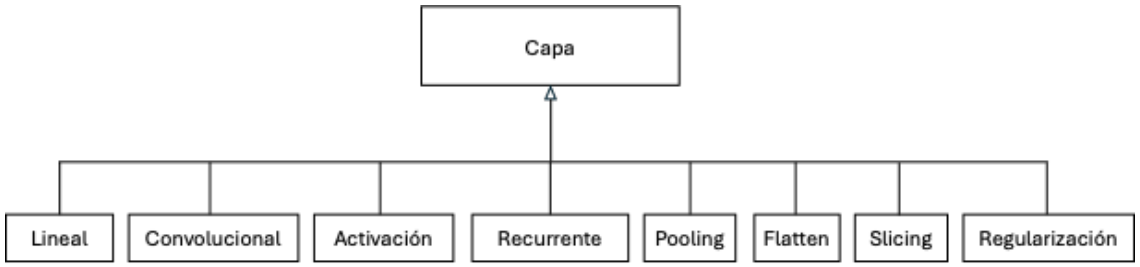

Figura 3.4: Diagrama de clases de las capas en UML.

3.3.1. Capa lineal

Conecta cada neurona de entrada con cada neurona de salida, lo que permite que cada neurona en la capa anterior afecte a cada neurona en la capa siguiente. La operación principal realizada por esta capa es una transformación lineal de los datos de entrada, donde cada neurona calcula una combinación lineal de las entradas ponderadas por sus respectivos pesos, a los que se suma un término de sesgo. Matemáticamente, esto se representa como la multiplicación de la matriz de entradas por una matriz de pesos y la adición del sesgo. Esta

capa es fundamental en la mayoría de las redes neuronales, ya que proporciona la capacidad de aprender representaciones no lineales de los datos.

3.3.2. Capa de activación

Aplican una función de activación a la salida de las neuronas en la red con el propósito de agregar no linealidad a la red neuronal. Esta adición de no linealidad es esencial ya que las funciones de activación introducen un paso adicional en cada capa durante la propagación directa. La función de activación determina si una neurona se activa o no, lo que a su vez decide la relevancia de la entrada de la neurona en el proceso de predicción, simplificando operaciones matemáticas. Su función principal es derivar la salida de un conjunto de valores de entrada proporcionados a un nodo, que es una réplica de una neurona que recibe una serie de señales de entrada. La tarea fundamental de la función de activación es transformar la suma ponderada de las entradas del nodo en un valor de salida para ser transmitido. Existe una amplia variedad de funciones de activación utilizadas en redes neuronales, cada una con características y aplicaciones específicas, que se encuentran desarrolladas en el anexo A.10.

3.3.3. Capa convolucional

La capa convolucional contiene un conjunto de núcleos convolucionales denominados *kernels*, que se aplican sobre la imagen de entrada para producir un mapa de características de salida. Como se puede apreciar en la figura 3.5, cada *kernel* consiste en una cuadrícula de valores discretos o numéricos, donde cada valor representa el peso asociado al núcleo. Durante la fase inicial del entrenamiento del modelo, los pesos de cada núcleo se inicializan de forma aleatoria. A medida que progresa el entrenamiento, estos pesos se ajustan para que el núcleo pueda aprender a extraer características relevantes de la entrada. Además de estos pesos, existen otros dos parámetros que configuran las capas convolucionales.

El parámetro de *stride* modifica la cantidad de desplazamiento del *kernel* sobre la entrada. Incrementar el *stride* resulta en una salida de menor tamaño.

Por otro lado, el *padding*, consiste en la cantidad de píxeles adicionales agregados alrededor de la imagen durante el proceso de convolución. Al aplicar un *kernel* sobre la imagen, esta tiende a reducirse en tamaño, lo que podría llevar a una pérdida de información en los bordes. Por lo tanto, el *padding* se utiliza para evitar esta reducción, agregando píxeles adicionales fuera de la imagen original. Esto garantiza que el tamaño de la imagen de salida se mantenga lo más cercano posible al tamaño original, compensando así la reducción causada por el paso. Además, el *padding* puede usarse para aumentar el tamaño de la imagen de entrada, lo que resulta en un mapa de características de salida de mayor tamaño.

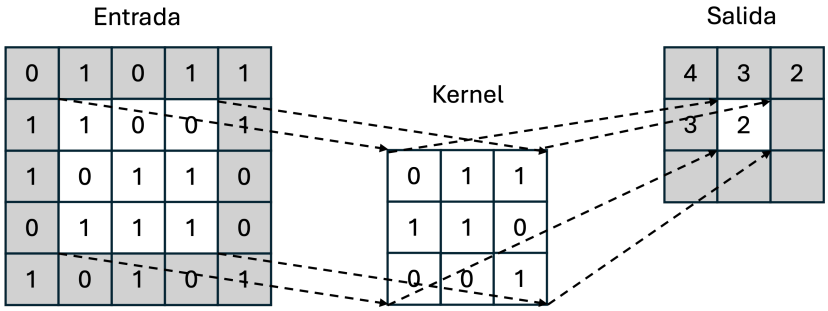

Figura 3.5: Funcionamiento de una capa convolucional.

3.3.4. Capa pooling

Reducen la dimensionalidad de los datos mediante la disminución del tamaño espacial de las representaciones, al tiempo que se conservan las características más relevantes. La operación de *pooling* se lleva a cabo al especificar el tamaño del *kernel* y el *stride* de la operación, de manera similar a la operación de convolución. Esta técnica contribuye a reducir tanto la cantidad de parámetros a aprender como la carga computacional de la red. La capa de *pooling* sintetiza las características presentes en una región del mapa de características generado por una capa de convolución, lo que conlleva a la realización de más operaciones sobre características resumidas en lugar de aquellas posicionadas con precisión por la capa convolucional. Este enfoque fortalece la capacidad del modelo para adaptarse a las variaciones en la posición de las características en la imagen de entrada.

Existen varios tipos de capas de *pooling* utilizadas comúmente, siendo las de la figura 3.6 las más utilizadas, cada una con sus propias características y aplicaciones específicas. El *max pooling* es una de los más comunes, donde se selecciona el valor máximo dentro de una región de la entrada, lo que ayuda a conservar las características más relevantes y a reducir la dimensionalidad de los datos. Por otro lado, el *average pooling* calcula el promedio de los valores dentro de la región de entrada, lo que puede ser útil cuando se desea una reducción suave de la dimensionalidad sin enfatizar características específicas. Además, existen variantes más sofisticadas como el *global pooling*, que calcula una sola salida para toda la característica, y el *adaptative pooling*, que adapta dinámicamente el tamaño de la región de pooling en función de las características presentes en la entrada.

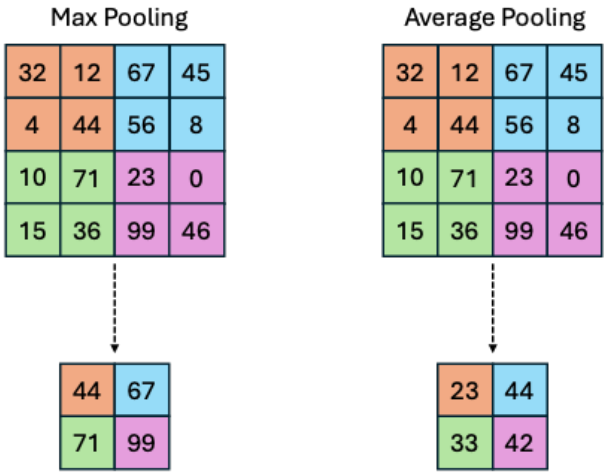

Figura 3.6: Funcionamiento de las capas *pooling*.

3.3.5. Capa flatten

Convierte los datos de entrada multidimensionales en un vector unidimensional, lo que facilita la conexión entre diferentes tipos de capas. Esta capa se sitúa típicamente entre las capas convolucionales o de *pooling* y las capas lineales, y su operación consiste en desplegar todas las características extraídas en un único vector de entrada. De esta manera, la capa *flatten* proporciona una transición suave entre la extracción de características en las primeras etapas de la red y la interpretación y toma de decisiones en las capas posteriores, lo que contribuye a la eficacia y eficiencia del modelo.

3.3.6. Capa recurrente

Las capas recurrentes están diseñadas para procesar datos secuenciales o de series temporales. Estas capas tienen la capacidad de mantener y utilizar información sobre secuencias anteriores de datos mientras procesan los datos actuales.

Las redes neuronales tradicionales permiten que los datos fluyan solo en una dirección, es decir, de entrada a salida, lo que las hace adecuadas para procesar datos con muestras que son independientes entre sí. Sin embargo, las recurrentes tienen señales que viajan en ambas direcciones mediante el uso de bucles de retroalimentación en la red. Las características derivadas de la entrada anterior se retroalimentan a la red, lo que les da la capacidad de memorizar y almacenar los estados o la información de las entradas anteriores para generar la siguiente salida de la secuencia. Adicionalmente, otra funcionalidad innovadora es la posibilidad de tener como entradas y salidas a secuencias de longitud variable.

Como se observa en la figura 3.7, la arquitectura de una capa recurrente consta de unidades recurrentes que reciben entradas de datos en cada paso de tiempo, así como una conexión recurrente, que permite que la salida de la capa en un paso de tiempo dado, se retroalimente

como entrada en el siguiente paso de tiempo. Esto permite que la capa recurrente mantenga un estado interno que encapsula la información relevante de pasos de tiempo anteriores. En las redes neuronales recurrentes ocurre un fenómeno conocido como *unfolding* que nos permite, en cada instante de tiempo, poder desplegar la red en pasos de tiempo para obtener la salida en el paso de tiempo siguiente. La red desplegada es muy similar a la red neuronal tradicional.

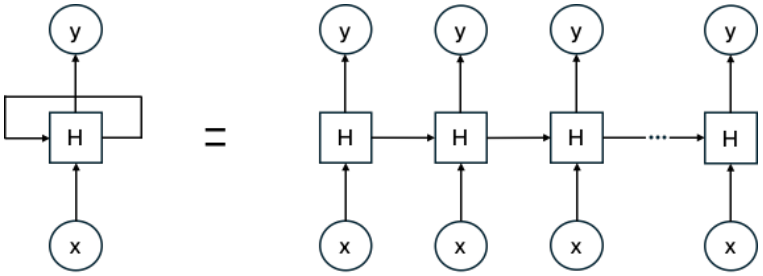

Figura 3.7: Funcionamiento de una capa recurrente en su forma normal y desenrollada.

Sin embargo, uno de los grandes problemas que pueden surgir con las redes neuronales recurrentes con la retroporpagación de los pesos es la cantidad de derivadas requeridas para una sola actualización de pesos en la actualización. Esto puede causar que los pesos se desvanezcan o exploten y que el aprendizaje se ralentice y la habilidad del modelo empeore. Esto hace que la retropropagación pueda ser computacionalmente costosa a medida que aumenta el número de pasos de tiempo. A partir de la idea inicial de capa recurrente, existen diferentes implementaciones que varían en su estructura y funcionalidad con el objetivo de superar estos problemas y obtener mejores rendimientos, las cuales se explican en el anexo B.2. Estas diferentes arquitecturas de capas recurrentes ofrecen a los desarrolladores opciones variadas para adaptarse a las características específicas de los datos y las necesidades de la tarea, contribuyendo así a un rendimiento óptimo del modelo en una amplia gama de aplicaciones.

Las capas recurrentes presentan diversos tipos de salida con un propósito específico cada una, fundamentales para el procesamiento y análisis de secuencias temporales, permitiendo elegir la salida deseada al definir una capa recurrente. Entre las opciones se encuentran la salida como tal, denominada *"EndSequece"*, que corresponde a la secuencia de valores emitidos en cada paso temporal, reflejando la transformación de las entradas a través de la capa. Otra elección son los estados ocultos, *"HiddenStates"*, los cuales encapsulan la información temporal acumulada hasta el paso actual y son esenciales para mantener la memoria a corto plazo de la secuencia. Por último, para algunos tipos de capa recurrente, se incluyen los estados de celda, *"CellStates"*, que almacenan información a más largo plazo, permitiendo a la red conservar y manipular datos importantes durante periodos extensos.

3.3.7. Capa slicing

La capa *slicing* se utiliza para seleccionar subsegmentos específicos de una secuencia, permitiendo un control más granular sobre los datos de salida. Al aceptar una secuencia

como entrada, esta capa permite definir un índice inicial y un índice final, determinando así el fragmento exacto de la secuencia que se desea extraer. Esta funcionalidad es especialmente útil en el procesamiento de secuencias largas, ya que facilita la focalización en subsecciones de interés, mejorando la eficiencia y precisión en el análisis y manipulación de los datos. De este modo, la capa slicing se convierte en una herramienta versátil para ajustar y optimizar el tratamiento de información secuencial.

3.3.8. Capa de regularización

Las redes neuronales se entrenan con un conjunto de datos finito de muestras. Es por ello que, una red neuronal entrenada debe ser evaluada experimentalmente con un conjunto de datos diferente al de entrenamiento con el objetivo de poder asegurar un rendimiento del modelo en la vida real. Si se observa la figura 3.8, el aprendizaje demasiado preciso sobre los datos de entrenamiento puede dar lugar a resultados pobres con datos nuevos, lo que se conoce como sobreajuste. Por otro lado, el aprendizaje poco preciso sobre los datos de entrenamiento puede dar lugar a también resultados pobres conocido como subajuste.

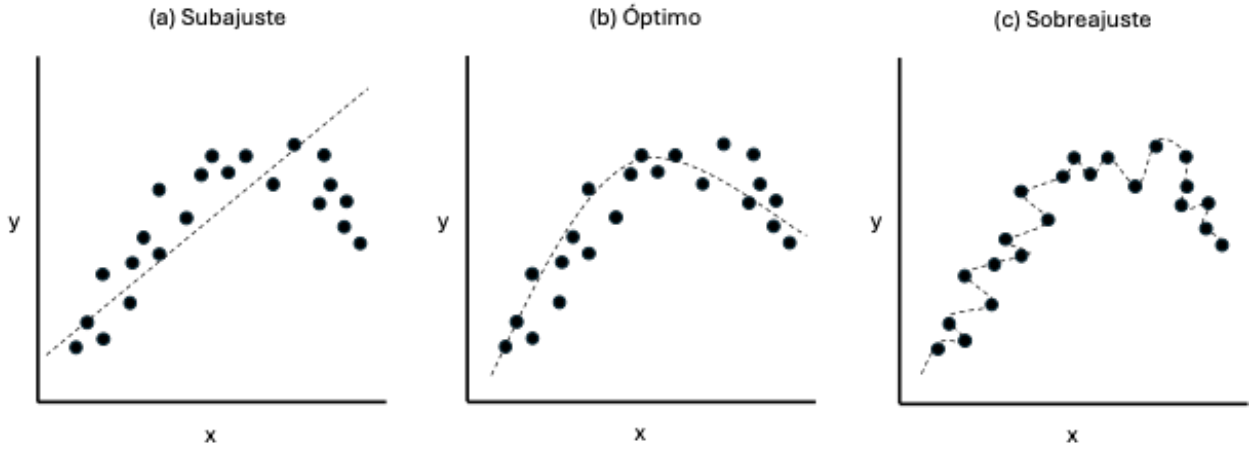

Figura 3.8: Problemas con el aprendizaje de redes neuronales.

La regularización se define como cualquier modificación aplicada a un algoritmo de aprendizaje con el objetivo de reducir su error de generalización y evitar problemas de sobreajuste y subajuste sin alterar su error de entrenamiento, mientras se mantiene una nivel de optimización. Existe una amplia variedad de funciones de activación utilizadas para ayudar a la generalización de la red neuronal que se explican en el anexo C.3. Estas técnicas de regularización ofrecen mecanismos efectivos para mejorar la generalización de los modelos de ML y evitar el sobreajuste en una variedad de tareas y arquitecturas de redes neuronales.

3.4. Bloque

Un nivel de abstracción por encima de las capas se encuentran los bloques. Un bloque es un conjunto organizado de capas que se agrupan para realizar una tarea específica. Este concepto surge debido a que, en las arquitecturas, comunmente se suelen repetir conjuntos de capas que siempre se encuentran en el mismo orden. Este enfoque permite una representación directa de las transformaciones aplicadas a los datos a medida que fluyen a través del bloque y ayuda a estructurar y modularizar la arquitectura de la red, facilitando su comprensión.

Los bloques están pensados para utilizarse de manera repetida y combinarse de diversas formas para construir arquitecturas de redes neuronales más complejas y efectivas para tareas específicas. La elección y configuración de los bloques en una red neuronal depende en gran medida de la naturaleza de los datos y de la tarea que se esté abordando.

Asimismo, el bloque sigue el patrón de diseño *facade*. Este patrón se utiliza para proporcionar una interfaz unificada a un conjunto de interfaces en un subsistema. En otras palabras, oculta la complejidad del sistema subyacente y proporciona una interfaz más simple para trabajar. De esta forma, cada nueva implementación de la clase bloque agrega alguna funcionalidad adicional al sistema, manteniendo la interfaz unificada para interactuar con el subsistema. Como se puede observar en la figura 3.9, hasta el momento sólo se han realizado dos implementaciones.

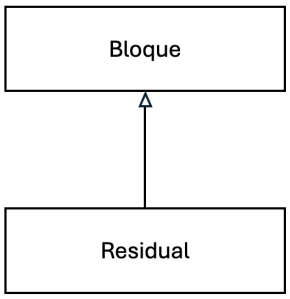

Figura 3.9: Diagrama de clases de los bloques en UML.

3.4.1. Bloque simple

Un bloque simple es un conjunto secuencial de capas que se aplican una tras otra para realizar una operación específica. Este tipo de bloque sigue una estructura lineal y directa, donde la salida de una capa se convierte en la entrada de la siguiente, sin conexiones adicionales que salten capas dentro del bloque. Los bloques simples son fundamentales en la construcción de redes neuronales debido a su simplicidad y efectividad en tareas específicas de procesamiento de datos. Al igual que los ladrillos construyen una pared, como se observa en la figura 3.10, los bloques simples se apilan y combinan para crear redes neuronales más complejas y capaces.

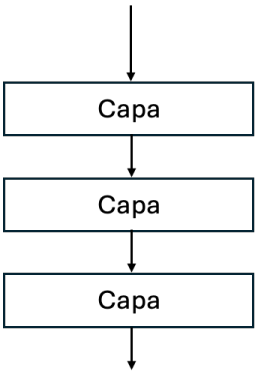

Figura 3.10: Estructura de un bloque simple.

3.4.2. Bloque residual

El propósito principal del bloque residual es facilitar el entrenamiento de redes neuronales profundas. La incorporación de un mayor número de capas en una red neuronal a menudo conlleva el desafío del desvanecimiento del gradiente, en el cual las actualizaciones de los pesos se hacen prácticamente inexistentes durante la propagación hacia atrás a través de la red durante el proceso de entrenamiento. Este fenómeno puede dificultar el entrenamiento eficiente de redes profundas y llevar a un rendimiento subóptimo.

La concepción del bloque residual se fundamenta en la suposición de que múltiples capas no lineales pueden aproximar asintóticamente funciones complejas, lo cual es equivalente a suponer que también pueden aproximar asintóticamente funciones residuales. La premisa clave detrás de estas conexiones residuales radica en que el bloque no solo se encarga de aprender la representación deseada de los datos de entrada, sino que también aprende la discrepancia entre dicha representación deseada y la entrada actual. El bloque residual incorpora un *shortcut* que traslada la entrada hasta el final del bloque, como se visualiza en la figura 3.11. Esta conexión se suma a la salida de las capas intermedias antes de ser transmitida a la siguiente capa. Esta estructura residual permite a la red neuronal aprender funciones de identidad, lo que simplifica el proceso de entrenamiento de redes más profundas y mitiga el problema del desvanecimiento del gradiente.

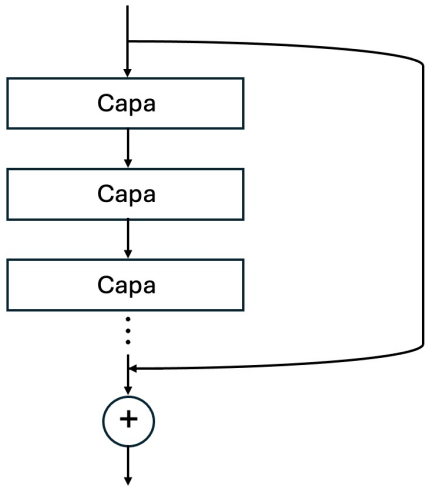

Figura 3.11: Estructura de un bloque residual.

3.5. Sección

Las secciones se corresponden con un conjunto de bloques que se ejecutan de forma secuencial para cumplir un objetivo específico en la red neuronal. De esta forma, las secciones permiten definir una tarea a realizar. Cada una de estas secciones tiene su propia estructura y función específicas, y puede ser adaptada y combinada de diversas formas para construir arquitecturas de redes neuronales más complejas y especializadas.

Las secciones actúan como componentes modulares de alto nivel dentro de la arquitectura de la red neuronal, permitiendo una organización clara y lógica de las distintas fases del procesamiento de datos. Al estructurar la red en secciones, se mejora la legibilidad y se facilita la implementación de cambios o mejoras. Además, las secciones proporcionan una forma de encapsular conjuntos de operaciones comunes, promoviendo la reutilización y la consistencia a lo largo de la arquitectura. Como se puede observar en la figura 3.12, existen tres tipos diferentes de sección.

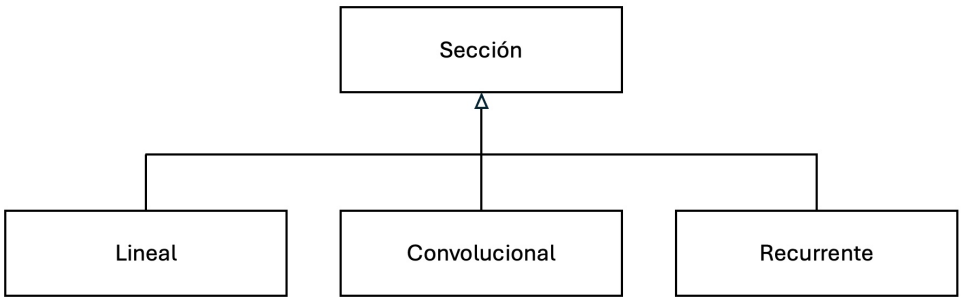

Figura 3.12: Diagrama de clases de las secciones en UML.

3.5.1. Sección lineal

Conjunto de bloques que contienen capas lineales, capas de regularización y capas de activación. Estas capas trabajan juntas para transformar las características de entrada de manera lineal, regularizar el modelo para prevenir el sobreajuste y aplicar no linealidad para capturar relaciones complejas en los datos.

La combinación de estas capas en una sección lineal permite que el modelo realice transformaciones complejas sobre diferentes tipos de datos de entrada, preparándolos para etapas posteriores de procesamiento o para la predicción final.

3.5.2. Sección convolucional

Conjunto de bloques que contienen capas convolucionales, capas *pooling*, capas de regularización y capas de activación. Estas capas trabajan juntas para extraer características relevantes de los datos de entrada, reducir la dimensionalidad, prevenir el sobreajuste y aplicar no linealidad para capturar relaciones complejas en los datos de entrada.

La sección convolucional es esencial en redes neuronales que procesan datos con estructura espacial, como imágenes, permitiendo la extracción de características jerárquicas a diferentes niveles de abstracción.

3.5.3. Sección recurrente

Conjunto de bloques que contienen capas recurrentes, capas de regularización y capas de activación. Estas capas trabajan en conjunto para procesar secuencias de datos en el tiempo, prevenir el sobreajuste y capturar relaciones no lineales en los datos secuenciales.

La sección recurrente es indispensable en aplicaciones que implican datos secuenciales, como series temporales, procesamiento de lenguaje natural y cualquier otra tarea donde el orden de los datos es relevante.

3.6. Implementación

Una vez que se han establecido los principios de diseño que sustentan el *framework* y se ha detallado su arquitectura en la fase de diseño, se procederá ahora a exponer su implementación con el fin de ilustrar de manera práctica cómo estos principios se aplican en la construcción real del sistema. Esta etapa de implementación permitirá demostrar cómo los conceptos teóricos se traducen en código concreto, brindando así una comprensión más completa de la funcionalidad y la eficacia del *framework* en la práctica.

La figura 3.13 muestra el uso de una clase abstracta para representar el constructo de arquitectura en el *framework* siguiendo el patrón de diseño *Fluent Interface*, el cual permite

encadenar llamadas de método de manera fluida para construir y manipular la arquitectura de manera expresiva y legible. Este enfoque mejora la claridad y la cohesión del código al proporcionar una sintaxis más natural y cercana al lenguaje humano para la definición de arquitecturas. Al crear una arquitectura, los desarrolladores pueden ir añadiendo componentes, tanto secciones como bloques o capas, usando el método "*attach*" de manera sucesiva, lo que facilita la construcción incremental de la arquitectura del modelo.

```python
class Architecture(ABC):
    def __init__(self, name: str):
        self.name = name

    @abstractmethod
    def attach(self, component: Union[Section, Block, Layer]):
        pass
```

Figura 3.13: Implementación del constructo de arquitectura en el *framework*.

Cada implementación concreta de la clase arquitectura, adaptada a las diferentes librerías subyacentes, siendo *Pytorch* la ejemplificada en la figura 3.14, demuestra cómo el diseño del *framework* se alinea con los principios expuestos. En primer lugar, al emplear la clase abstracta de arquitectura, se promueve el SRP al encapsular la funcionalidad relacionada con la definición y manipulación de la arquitectura del modelo en una sola entidad cohesiva. Esta clase abstracta se enfoca exclusivamente en la representación y manipulación de la arquitectura, lo que facilita su comprensión, mantenimiento y extensión. Además de eso, al depender de la clase abstracta en lugar de la implemetación específica, permite cambiar la implementación concreta sin alterar el código que depende de la abstracción, cumpliendo de esta forma el ISP.

Por otro lado, la implementación concreta de arquitectura usando la librería *Pytorch* refleja el OCP estando abierta para la extensión, ya que permite la adición de componentes específicos de *Pytorch* sin modificar su código fuente original. Al mismo tiempo, permanece cerrada para la modificación, ya que los cambios en la funcionalidad de la arquitectura pueden realizarse mediante la extensión de esta clase sin alterar su comportamiento existente. Adicionalmente, el DIP se ve reflejado al definir una interfaz común para representar arquitecturas y permitir la adición de componentes específicos de cada biblioteca subyacente, reduciendo el acoplamiento entre el *framework* y las implementaciones concretas. Esto facilita la sustitución de las implementaciones subyacentes sin afectar otras partes del sistema.

```python
class PytorchArchitecture(Module, Architecture):
    def __init__(self, name: str):
        Module.__init__(self)
        Architecture.__init__(self, name)
        self.components = Sequential()

    def attach(self, component: Union[PytorchSection, PytorchBlock, PytorchLayer]):
        self.components.append(component)
        return self

    def forward(self, x):
        return self.components(x)
```

Figura 3.14: Implementación específica de *Pytorch* del constructo de arquitectura en el *framework*.

Este enfoque se aplica en todos los constructos del *framework*, como son los de sección, bloque y capa. Para cada uno de estos elementos, se define una clase abstracta que encapsula la funcionalidad común, seguida de implementaciones concretas específicas para cada librería. De esta forma, se asegura que el *framework* sea extensible y adaptable a diferentes tecnologías subyacentes sin comprometer su diseño modular y cohesivo.

En cuanto al LSP, este principio se demuestra claramente a través de la jerarquía de capas diferentes definidas. Así, la implementación genérica de capa que se muestra en la figura 3.15 se puede derivar en una clase más específica, como es la de la capa lineal, que se puede observar en la figura 3.16, existiendo su implementación en *Pytorch* como se demuestra en la figura 3.17. Gracias a esta estructura jerárquica, una capa lineal de *Pytorch* puede ser utilizada en cualquier lugar donde se espera una capa, pudiendo usarse cualquier tipo de capa de las existentes en el *framework*, cumpliendo así con el LSP.

```python
class Layer:
    pass
```

Figura 3.15: Implementación del constructo de capa en el *framework*.

```python
class LinearLayer(Layer):
    pass
```

Figura 3.16: Implementación del constructo de capa lineal en el *framework*.

```
class PytorchLinearLayer(PytorchLayer, LinearLayer):
    def __init__(self, in_features: int, out_features: int, dimension: int = -1, bias: bool = True):
        super(PytorchLinearLayer, self).__init__()
        self.dimension = dimension
        self.layer = nn.Linear(in_features=in_features, out_features=out_features, bias=bias)

    def forward(self, x: Tensor) -> Tensor:
        if self.dimension == -1:
            return self.layer(x)
        return torch.transpose(self.layer(torch.transpose(x, self.dimension, -1)), self.dimension, -1)
```

Figura 3.17: Implementación específica de *Pytorch* del constructo de capa lineal en el *framework*.

Capítulo 4

Integración

En este capítulo, se presenta el proceso de integración del *framework* estructural con el resto de componentes del sistema. Por un lado, se expondrá la integración con el DSL, la cual requiere una serie de componentes fundamentales, tales como el generador de código y las plantillas, que facilitan la transformación automática de las abstracciones de alto nivel en código concreto. Se explorarán los detalles del generador de código, se describirá el motor de plantillas y se mostrarán las plantillas creadas. Por otro lado, se expondrá la integración con el *framework* operacional explicando el uso que se hace de las arquitecturas en el mismo.

4.1. Framework estructural con DSL

La integración con el DSL requiere de una serie de componentes para su realización, como son el generador de código y plantillas, permitiendo una transformación fluida y automática de las abstracciones de alto nivel del DSL en código concreto del *framework* gracias a su actuación conjunta.

4.1.1. Generador de código

El generador de código es fundamental en el sistema debido a que permite la traducción de las especificaciones definidas en el DSL a código ejecutable en el *framework*. La necesidad de este componente radica en la automatización y estandarización del proceso de desarrollo, permitiendo convertir de manera sistemática y precisa las abstracciones de alto nivel en implementaciones concretas. Este generador asegura que las especificaciones del DSL se interpreten y traduzcan de manera uniforme, reduciendo la posibilidad de errores manuales y mejorando la eficiencia del desarrollo. Además, facilita la mantenibilidad del software al permitir que los cambios en las especificaciones del DSL se reflejen automáticamente en el código generado, promoviendo una mayor coherencia y alineación con los requisitos del dominio.

El diseño del *framework* se ha concebido de manera homomorfa al del DSL, lo que implica que existe una correspondencia directa y natural entre los constructos del DSL y los del *framework*. Esta correspondencia asegura que cada componente y estructura definida en el DSL tenga una representación clara y equivalente en el *framework*, facilitando así la generación automática de código. Para lograr esto, ha sido crucial mantener una cohesión estrecha entre la definición en el DSL y lo generado en el *framework*. Cada constructo del DSL, ya sea una sección, bloque o capa, tiene su representación correspondiente en el *framework*, asegurando que las funcionalidades y relaciones especificadas en el DSL se reflejen de manera exacta en la implementación final. Esta cohesión no solo mejora la mantenibilidad del sistema, sino que también garantiza la precisión y robustez de la arquitectura generada.

Durante el desarrollo, surgieron discrepancias entre la definición del DSL y del *framework*, que tuvieron que ser resueltas mediante soluciones mutuas. Por ejemplo, en varios casos, las características deseadas para los constructos del DSL no correspondían exactamente con las implementaciones previstas en el *framework*, lo que obligó a ajustar y armonizar ambos diseños. Estas discrepancias se resolvieron mediante un análisis detallado y discusiones colaborativas, asegurando que tanto el DSL como el *framework* cumplieran con los principios de diseño y funcionalidad esperados. Este proceso, aunque desafiante, ha resultado en un sistema más cohesivo y robusto, garantizando que la arquitectura generada desde el DSL sea fiel a los conceptos y estructuras definidos en el *framework*.

4.1.2. Motor de plantillas

Para la generación de código se ha utilizado un motor de plantillas, también conocido como motor de reglas, el cual proporciona un modelo computacional alternativo al modelo imperativo habitual que se basa en comandos secuenciales con condicionales y bucles. En lugar de esto, un motor de plantillas utiliza un sistema de reglas de producción. Este sistema se compone de un conjunto de reglas, cada una de las cuales define una condición y una acción. La clave está en que las reglas pueden ser definidas en cualquier orden, y es el motor de plantillas el que decide cuándo y en qué orden evaluarlas para generar el código final de manera lógica y eficiente.

Por ejemplo, en un motor de reglas, el sistema recorre todas las reglas, selecciona aquellas cuya condición es verdadera y evalúa las acciones correspondientes. Esta metodología es efectiva para diversos problemas que se ajustan naturalmente a este modelo, permitiendo una programación más flexible y modular. Del mismo modo, un motor de plantillas recorre las definiciones del DSL, aplica las plantillas correspondientes basadas en las condiciones especificadas y genera el código necesario para el *framework*.

La utilización de un motor de plantillas en la generación de código no solo agiliza el proceso de desarrollo, sino que también asegura la consistencia y precisión del código generado, reduciendo errores y facilitando la mantenibilidad del software. Además, permite la integración del DSL con los *framework* desarrollados, aprovechando las ventajas del modelo computacional de reglas de producción para abordar de manera efectiva las necesidades específicas del dominio del proyecto. Esto significa que, si en algún momento se decide cambiar

el *framework* utilizado o modificar las especificaciones del mismo, el motor de plantillas puede ajustarse rápidamente para generar el nuevo código requerido, manteniendo la integridad y consistencia del proyecto.

La utilización de plantillas ofrece una abstracción que separa las reglas de negocio y la lógica específica del dominio del código concreto del *framework*. Al hacerlo, se asegura que las reglas definidas en el DSL sean interpretadas y ejecutadas correctamente en cualquier entorno de desarrollo. Este enfoque no solo incrementa la modularidad y mantenibilidad del código, sino que también facilita la adaptación a nuevos requerimientos o tecnologías sin una reestructuración significativa del proyecto.

En las figuras referentes a plantillas, para mantener el código dentro de un ancho manejable, se ha utilizado el carácter "\", que se utiliza en Python para continuar una línea de código en la siguiente línea, ayudando de esta forma a mejorar la legibilidad para el lector.

La figura 4.1 muestra la estructura de una regla, definida como "si x entonces y", es decir, cuando se cumple una condición se realiza una acción. Las condiciones se expresan como un conjunto de funciones booleanas mientras que las acciones contienen el texto que será renderizado. Cuando las condiciones de las reglas se cumplen, se dice que la regla se ha disparado, y su acción será ejecutada.

```
def type(layer)
    $type+CamelCase~Layer()
end
```

Figura 4.1: Regla genérica para renderizar una capa con el motor de plantillas.

De está forma, el uso del término "def" indica que se está definiendo una nueva regla, y la condición "type(Layer)" especifica que esta regla se ejecutará si el objeto activador es de tipo *Layer*. La acción que se ejecutará es la expresión "$type+CamelCase~Layer()", donde "$type" es una marca empleada para identificar un atributo del objeto que desencadenará otras reglas, en este caso escribir el tipo de capa. El formateador "+CamelCase" determina cómo se renderizarán estos objetos, en este caso siendo *Camel Case*. Finalmente, la palabra clave "end" señala el final de la regla.

Sin embargo, diferentes tipos de capas requieren una generación de diferente forma debido a que contienen parámetros específicos. Para ello, como se observa en la figura 4.2, se puede usar el operador "&" para indicar que un objeto debe tener todos los tipos especificados. En este caso, la generación de una capa lineal se diferencia de las demás capas en que necesita una serie de parámetros solo existentes en la misma. Adicionalmente, cuando varios objetos contienen los mismos parámetros pero no se renderizan de una forma genérica tampoco, se puede usar el operador "|" para indicar que un objeto debe tener cualquiera de los tipos especificados. En la figura 4.3 se muestra un ejemplo donde ambas capas *pooling*, *maxpool* y *avgpool*, se renderizan de la misma forma pero su código es diferente al de la capa genérica.

```
def type(Linear & layer)
    $type+CamelCase~Layer(in_features=$in_features, out_features=$out_features,\
                          dimension=$dimension, bias=$bias)
end
```

Figura 4.2: Regla para renderizar una capa lineal con el motor de plantillas.

```
def type(MaxPool | AvgPool) type(layer)
    $type+CamelCase~Layer(kernel=$kernel, stride=$stride, padding=$padding)
end
```

Figura 4.3: Regla para renderizar una capa *pooling* con el motor de plantillas.

Otra forma de lanzar una regla es con la función *trigger*. Cuando una regla tiene una condición de disparo significa que se activará cuando la marca que se haya especificado este existente o el usuario haya establecido que se realice. En la figura 4.4 se puede observar la aplicabilidad que tiene esta condición para renderizar la definición para importar una capa genérica. Como se vió anteriormente, ya existe una regla definida para una capa genérica que se encarga de instanciar la capa, sin embargo, en este caso, el objetivo es diferente por lo que se usa el disparador para renderizar la definición de importar la clase.

```
def type(layer) trigger(import)
    from implementations.$library.architecture.layers.$type+Lowercase \
        import $library+FirstUppercase~$type~Layer as $type~Layer
end
```

Figura 4.4: Regla para renderizar la definición de importar una capa genérica con el motor de plantillas.

Existe otra forma de lanzar una regla que es con la condición de atributo donde un objeto solo se renderiza si contiene un atributo específico. La figura 4.5 demuestra la aplicabilidad de esta función puesto que, en casos donde la capa se encuentre en un paquete específico, no será posible usar la regla definida anteriormente ya que dicho atributo no se tendría en cuenta. En cambio, haciendo uso de dicha condición, se puede crear una regla que permita especificar el paquete desde el cual se quiere importar la capa específica.

```
def type(layer) trigger(import) attribute(package)
    from implementations.$library.architecture.layers.$package.$type+Lowercase \
        import $library+FirstUppercase~$type~Layer as $type~Layer
end
```

Figura 4.5: Regla para renderizar la definición de importar una capa contenida en un paquete específico con el motor de plantillas.

Por último, existe el caso en que se quieran renderizar múltiples objetos contenidos en una estructura de almacenamiento como puede ser una lista. Para ese tipo de casos, "..." está

reservado como símbolo del sistema para identificar dicho tipo de casos y renderizar todos los objetos que contiene, el cual debe ir seguido de un abrir y cerrar corchetes indicando el carácter que se usará para separar los valores. En este caso, "NL" indica que se renderizará cada objeto en una linea diferente. Este comportamiento se puede ver reflejado en la figura 4.6 donde se realizan las definiciones de importar todas las capas contenidas en la arquitectura.

```
def type(import)
    $layer+import...[$NL]
end
```

Figura 4.6: Regla para renderizar las definiciones de importar las capas de una arquitectura con el motor de plantillas.

Una vez las reglas se encuentran definidas, el motor de plantillas usa un objeto de *Java* para lanzar las reglas correspondientes. Una regla se dispara cuando este disparador hace que todas las funciones de sus condiciones sean verdaderas. Cuando se pueden disparar varias reglas, como las reglas están ordenadas, se ejecuta la primera regla que coincida con el disparador actual, por lo que el orden de definición de las reglas en la plantilla es crucial para su correcto funcionamiento.

De esta forma, es necesario convertir el objeto arquitectura producido por DSL a un objeto arquitectura del motor de plantillas capaz de ejecutar las reglas necesarias que permitan una correcta renderización del código del *framework* estructural.

4.1.3. Plantillas

Para permitir la materialización de una arquitectura en el *framework* estructural, es necesario convertir la clase "ArchitectureView", generada por el DSL a partir de la definición realizada por un usuario, en un "FrameBuilder", que es la clase que contiene los elementos necesarios para renderizar una plantilla. Esta conversión es esencial, ya que el "FrameBuilder" actúa como un contenedor estructurado encargado de organizar y encapsular los diversos elementos de la arquitectura, tales como secciones, bloques y capas, facilitando su procesamiento y renderización por el sistema.

Una vez dicha conversión está realizada, se ha de ejecutar la plantilla con las reglas encargadas de transformar dicha arquitectura en una arquitectura ejecutable por parte del *framework* estructural. Para ello, la plantilla ha de tener definidas tanto las reglas encargadas de generar las definiciones de importar las clases como de crear la instancia de arquitectura correspondiente. De esta forma, el punto de entrada inicial será el que se observa en la figura 4.7, que define una regla que se encargue de llamar sucesivamente al resto de reglas que se encargan de lo antes mencionado además de establecer el nombre que tendrá la arquiectura en cuestión.

```
def type(architecture)
    from implementations.$library.architecture.architecture \
        import $library+FirstUppercase~Architecture as Architecture
    $import

    architecture = (Architecture("$name")
                        $component...[$NL])
end
```

Figura 4.7: Regla para renderizar la arquitectura definida en el DSL con el motor de plantillas.

En primer lugar, como se observa en la figura 4.8 se renderizarán las declaraciones de los módulos necesarios para poder usar las clases pertinentes para el correcto functionamiento, recorriendo todos los componentes de la arquitectura, tanto secciones, como bloques y capas.

```
def type(import)
    $section+import...[$NL]
    $block+import...[$NL]
    $layer+import...[$NL]
end
```

Figura 4.8: Regla para renderizar las definiciones de importar las clases necesarias con el motor de plantillas.

Una vez importadas todas las clases, se comenzará renderizando la arquitectura, pudiendo renderizar tanto secciones como capas de forma aislada como se observa en la figura 4.9.

```
def type(component) attribute(section)
    .attach($section)
end

def type(component) attribute(layer)
    .attach($layer)
end
```

Figura 4.9: Regla para renderizar los componentes de la arquitectura con el motor de plantillas.

En caso de ser una capa, su renderización será directa dependiendo del tipo que sea. En el caso de las secciones, se ha de renderizar todo el conjunto de bloques que conforman la misma, tal y como se aprecia en la regla de la figura 4.10. Los bloques pueden ser renderizados de forma genérica o, en caso de ser de un tipo concreto, especificando el tipo como se observa en la figura 4.11.

```
def type(section)
    $type+CamelCase~Section($[
                $block...[, $NL]
    $])
end
```

Figura 4.10: Regla para renderizar una sección de la arquitectura con el motor de plantillas.

```
def type(block) attribute(type)
    $type+CamelCase~Block($[
        $layer...[, $NL]
    $])
end

def type(block)
    Block($[
        $layer...[, $NL]
    $])
end
```

Figura 4.11: Regla para renderizar un bloque de la arquitectura con el motor de plantillas.

4.2. Framework estructural con framework operacional

La integración entre el *framework* estructural y el *framework* operacional es esencial para gestionar de manera eficiente el ciclo de vida de una red neuronal, desde su definición hasta su ejecución y evaluación. El *framework* estructural se encarga de la especificación y organización de la arquitectura de la red neuronal, proporcionando una representación clara y modular de sus componentes, tales como capas, secciones y bloques. Esta estructura, una vez definida y validada, se transfiere al *framework* operacional, que maneja todos los aspectos relacionados con el entrenamiento, ajuste y despliegue de la red.

En el *framework* operacional, los constructos de laboratorio y experimento son esenciales para la organización y ejecución de pruebas con redes neuronales. El laboratorio representa el entorno controlado donde se llevan a cabo las diversas pruebas y estudios. Es el espacio dedicado a la configuración, ejecución y monitoreo de múltiples experimentos, proporcionando los recursos necesarios para el desarrollo y evaluación de modelos de DL. Por analogía con la vida real, el laboratorio es similar a un centro de investigación equipado con herramientas y tecnología avanzada para realizar investigaciones.

Cada experimento, por su parte, se refiere a la configuración específica usada para entrenar una red neuronal dentro del laboratorio. Define detalladamente la arquitectura a utilizar, la función de pérdida que se utilizará para evaluar el desempeño del modelo, el optimizador seleccionado para ajustar los pesos y demás parámetros que guiarán el proceso de entrenamiento. En términos prácticos, un experimento puede ser comparado con un proyecto

individual de investigación dentro de un laboratorio científico, donde se prueban hipótesis específicas bajo condiciones controladas y se analizan los resultados obtenidos para extraer conclusiones válidas. Esta estructuración permite una gestión meticulosa y reproducible de los estudios realizados en el ámbito de las redes neuronales.

Adicionalmente, el laboratorio tiene la responsabilidad de registrar los resultados de cada experimento de manera meticulosa. Finalmente, el laboratorio se encarga de identificar y devolver el modelo del experimento que ha obtenido el mejor rendimiento, preparándolo para su puesta en producción. Este proceso asegura que solo los modelos más eficaces y optimizados sean implementados en aplicaciones prácticas, garantizando así la calidad y fiabilidad de las soluciones desarrolladas

De esta forma, el *framework* estructural tiene la responsabilidad de inyectar las redes neuronales diseñadas en los experimentos pertinentes del *framework* operacional. Para ello, se ha definido un método en las arquitecturas que permite realizar el paso de los datos a través de la red neuronal. Este método asegura que el entrenamiento se pueda llevar a cabo de manera efectiva, encapsulando toda la lógica respectiva a la arquitectura.

Esta integración es esencial para que los datos de entrada sean procesados correctamente por la red neuronal durante el entrenamiento, permitiendo activar las funciones de pérdida y los optimizadores especificados durante el entrenamiento. Así, se garantiza que cada experimento utilice la red neuronal correspondiente y que el entrenamiento se ejecute de manera eficiente y precisa. La colaboración entre los *frameworks* estructural y operacional permite una evaluación continua y optimización de los modelos, asegurando resultados consistentes y reproducibles.

Capítulo 5

Ejemplos de uso

En este capítulo, se presentan ejemplos prácticos que ilustran el funcionamiento y las capacidades del proyecto de fin de grado desarrollado mostrando la forma de implementar diferentes tipos de redes neuronales.

5.1. Red neuronal artificial

La ANN, como se ha expuesto anteriormente, es una de las redes más simples de implementar, por lo que empezar por implementar la misma sentará las bases para el resto de implementaciones. La ANN puede ser utilizada para una variedad de tareas como pueden ser clasificación y regresión. En este caso, se diseñará un modelo específico de regresión.

La figura 5.1 muestra la implementación de una ANN. Para comenzar a definir una arquitectura, es necesario definir la clase "Architecture", la cual puede tener asignado un nombre de arquitectura, siendo en este caso "RegressiveANN". Una vez instanciada la clase arquitectura, se pueden añadir componentes usando el método "*attach*". En este caso, se le añade una sección lineal, siendo este tipo de secciones las usadas para una ANN. Esta sección ha de contener una lista de bloques definidos con la clase "Block", encontrando en este caso tres bloques. Cada bloque ha de contener una lista de capas que realicen una tarea concreta.

```
(Architecture("RegressiveANN")
     .attach(LinearSection([
                 Block([
                     LinearLayer(in_features=11, out_features=30, dimension=-1, bias=True),
                     BatchNormalizationLayer(num_features=30, eps=1.0E-5, momentum=0.3),
                     ReLULayer(),
                     DropoutLayer(probability=0.5)
                 ]),
                 Block([
                     LinearLayer(in_features=30, out_features=10, dimension=-1, bias=True),
                     BatchNormalizationLayer(num_features=10, eps=1.0E-5, momentum=0.5),
                     ReLULayer(),
                     DropoutLayer(probability=0.4)
                 ]),
                 Block([
                     LinearLayer(in_features=10, out_features=1, dimension=-1, bias=True),
                     ReLULayer()
                 ])
     ]))
```

Figura 5.1: Red neuronal artificial implementada en el *framework*.

Una de las capas usadas en esta arquitectura es la lineal, definida en la clase “LinearLayer”. Esta capa acepta como parámetros el tamaño de entrada “in_features”, el número de neuronas de la capa “out_features”, la dimensión de los datos en la que se quiere aplicar la linealidad “dimension” y una variable *booleana* que indica si quiere añadirse un sesgo a dicha capa “bias”. En cuanto al argumento de dimensión, este tiene como valor por defecto -1, indicando que se realiazará en la primera dimensión de los datos de entrada.

Otras de las capas definidas en la arquitectura son las de regularización, en este caso la de normalización de lotes “BatchNormalization”. Esta capa tiene como parámetros el tamaño de entrada “num_features”, un valor añadido al denominador para la estabilidad numérica “eps” y el valor utilizado para el cálculo de la media y la varianza “momentum”. Adicionalmente, la arquitectura contiene otra capa de regularización como es la capa de abandono, “Dropout”, que acepta como parámetro la probabilidad de que una neurona se desactive.

Por último, se encuentra una capa de activación, “ReLULayer”, sin embargo, se podría utilizar cualquier capa de activación deseada de las expuestas en el anexo A.10.

Una vez definidas las diferentes capas posibles que se pueden definir en la ANN, si se analiza la estructura de la implementación realizada, contiene una sección lineal que está formada por tres bloques simples. En los dos primeros bloques, se aplican de forma secuencial una linealidad, normalización de lotes para normalizar las activaciones de las neuronas, una función de activación y se finaliza con abandono de neuronas para prevenir la coadaptación de las neuronas y a mejorar la generalización del modelo. Así, en el primer bloque se aumenta la dimensionalidad de los datos, pasando de un vector de características de tamaño 11 a uno de 30, volviendo a reducir posteriormente dicho tamaño en el segundo bloque, acabando con un vector de características de tamaño 10. Por último, el tercer bloque aplica una linealidad seguida de una activación que convierte el vector de tamaño 10 en un único valor de salida, cumpliendo con la tarea de regresión.

5.2. Red neuronal convolucional

La CNN es una arquitectura fundamental en el campo del DL, especialmente en tareas de visión por computador. Al igual que la ANN, la CNN puede utilizarse para una variedad de tareas, incluyendo clasificación, detección de objetos y segmentación semántica. En este caso, se desarrollará un modelo específico para la clasificación de imágenes.

La figura 5.2 muestra la implementación de una CNN. Para definir una arquitectura, es necesario definir la clase "Architecture", como se ha comentado anteriormente, siendo en este caso el nombre "ClassificationCNN". Una vez instanciada la clase arquitectura, se le añade una sección convolucional, siendo este tipo de secciones las principales para una CNN. Esta sección ha de contener una lista de bloques, definidos con la clase "Block", encontrando en este caso dos bloques.

```
(Architecture("ClassificationCNN")
     .attach(ConvolutionalSection([
        Block([
           ConvolutionalLayer(in_channels=3, out_channels=33, kernel=(3, 3), stride=(2, 2), padding=(0, 0)),
           LogSigmoidLayer(),
           BatchNormalizationLayer(num_features=33, eps=1.0E-5, momentum=0.1),
           MaxPoolLayer(kernel=(5, 5), stride=(4, 4), padding=(0, 0))
         ]),
         Block([
           ConvolutionalLayer(in_channels=33, out_channels=16, kernel=(3, 3), stride=(3, 3), padding=(0, 0)),
           LogSigmoidLayer(),
           BatchNormalizationLayer(num_features=16, eps=1.0E-5, momentum=0.1),
           MaxPoolLayer(kernel=(3, 3), stride=(2, 2), padding=(1, 1))
         ])
     ]))
     .attach(FlattenLayer(from_dim=3, to_dim=1))
     .attach(LinearSection([
                Block([
                    LinearLayer(in_features=64, out_features=2, dimension=-1, bias=True),
                    SoftmaxLayer(dimension=-1)
                ])
     ])))
```

Figura 5.2: Red neuronal convolucional implementada en el *framework*.

Como novedad en esta arquitectura, se incluyen una serie de capas específicas para operaciones convolucionales. Por un lado, se introduce la capa convolucional, definida en la clase "ConvolutionalLayer". Esta capa acepta varios parámetros. El primero es 'in_channels", que indica el número de canales de entrada necesarios para arquitecturas con entradas multidimensionales. Luego está "out_channels", que define el número de canales de salida. El parámetro "kernel" especifica el tamaño de la matriz usada como *kernel*; puede ser un número, para definir una matriz cuadrada, o una tupla, para una matriz específica. "stride" se refiere al salto aplicado en las dimensiones x e y; puede ser un número para el mismo paso en ambas dimensiones o una tupla para definir pasos específicos por dimensión. Finalmente, "padding" define el tamaño del relleno aplicado en las dimensiones x e y, y también puede ser un número para el mismo relleno en ambas dimensiones o una tupla para definir rellenos específicos por dimensión.

La otra capa específica de esta arquitectura es la capa *pooling* que como parámetros acepta *kernel*, *stride* y *padding* que se definen igual que en la capa convolucional. El tipo de *pooling* a realizar puede ser elegido, aplicando en este caso concreto un *max pool*.

Posterior a la sección convolucional, se incluye una capa *flatten* cuyo propósito es aplanar las dimensiones con el objetivo de unir la sección convolucional con secciones lineales posteriores que realicen la clasificación. La capa *flatten* acepta como parámetros el número de dimensiones de entrada "from_dim" y el número de dimensiones que se quieren obtener como salida "to_dim", aunque su comportamiento por defecto cuando no se definen parámetros es aplanar todas las dimensiones en una única.

Una vez definidas las diferentes capas posibles que se pueden definir en la CNN, si se analiza la estructura de la implementación realizada, contiene una sección convolucional inicial que permite extraer las características de las imágenes, que está formada por dos bloques simples. En los dos primeros bloques, se aplican de forma secuencial una convolución, una función de activación, normalización de lotes y se finaliza con *pooling* para agrupar las características encontradas. Así, en el primer bloque se aumenta la dimensionalidad de los datos, pasando de una imagen en 3 dimensiones a una con 33, volviendo a reducir posteriormente dicho tamaño en el segundo bloque, acabando con 16 dimensiones. Seguidamente, se aplanan todas las dimensiones, obteniendo un vector con todas las características. Por último, se pasan dichas características por una sección lineal que contiene un bloque encargado de aplicar una linealidad que devuelve un vector de tamaño 2 (número de categorías) seguida de una *Softmax* que permite pasar ese vector a probabilidades, cumpliendo con la tarea de clasificación.

El uso de la capa *Softmax* es crucial para la tarea de clasificación ya que se encarga de convertir un vector de números reales en una distribución de probabilidad, donde cada valor del vector indica la probabilidad de pertenencia a cada clase. Esta capa es la única capa de activación que tiene un argumento, "dimension", que indica en qué dimensión se quiere aplicar el *softmax* cuyo valor por defecto es -1 indicando que se realizará en la última dimensión.

5.3. Red neuronal recurrente

La RNN es una arquitectura fundamental en tareas de procesamiento de secuencias, como el procesamiento de lenguaje natural y la traducción automática. En este caso, se procederá con el diseño de un modelo específico para la clasificación de secuencias.

La figura 5.3 muestra la implementación de una RNN. Para definir una arquitectura, es necesario definir la clase "Architecture", como se ha comentado anteriormente, siendo en este caso el nombre "ClassificationRNN". Una vez instanciada la clase arquitectura, se le añade una sección recurrente, siendo este tipo de secciones las principales para una RNN. Esta sección ha de contener una lista de bloques, definidos con la clase "Block", encontrando en este caso un único bloque.

```
(Architecture("ClassificationRNN")
     .attach(RecurrentSection([
        Block([
          RNNLayer(input_size=28, hidden_size=512, output_type=EndSequence, num_layer=4, bidirectional=False, dropout=0.0),
          FlattenLayer(from_dim=2, to_dim=1),
          ReLULayer()
        ]),
        Block([
          LinearLayer(in_features=14336, out_features=300, dimension=-1, bias=True),
          ReLULayer()
        ]),
        Block([
          LinearLayer(in_features=300, out_features=75, dimension=-1, bias=True),
          ReLULayer()
        ]),
        Block([
          LinearLayer(in_features=75, out_features=10, dimension=-1, bias=True),
          SoftmaxLayer(dimension=-1)
        ])
 ])))
```

Figura 5.3: Red neuronal recurrente implementada en el *framework*.

La sección recurrente es la parte central de la arquitectura. Aquí se puede utilizar cualquier tipo de capa recurrente, usando en este caso la básica con la clase "RNNLayer". Las capas recurrentes toman como parámetros el tamaño de entrada "input_size", el tamaño de la capa oculta "hidden_size", el tipo de salida a elegir "output_type", el número de capas "num_layer", una variable *booleana* que indica si es bidireccional o no "bidirectional", y la tasa de abandono "dropout". La salida de esta capa se pasa a través de una capa *flatten* para convertirla en un vector unidimensional, seguida de una capa de activación.

Después de la sección recurrente, las características aplanadas se pasan a través de una serie de bloques lineales para la clasificación final. Cada bloque consta de una capa lineal seguida de una capa de activación. En este caso, hay tres bloques lineales, donde cada uno reduce gradualmente el tamaño de las características hasta llegar al número de clases de salida, siendo en este caso 10. Por último, la salida de la última capa lineal se pasa a través de una capa *softmax* para obtener las probabilidades de clasificación de las clases.

Capítulo 6

Desarrollo

En este capítulo se describe la metodología y cronología seguidas durante el desarrollo del trabajo de fin de grado. Por otro lado, se describe el marco tecnológico del trabajo, el cual proporciona el contexto y la infraestructura además de describir las herramientas y tecnologías utilizadas para el desarrollo y la implementación del *framework* desarrollado.

6.1. Trabajo previo

Los conocimientos adquiridos a lo largo de la carrera han proporcionado una base sólida y esencial para la elaboración del trabajo de fin de grado. Cada asignatura cursada ha contribuido de manera significativa al desarrollo de habilidades técnicas y teóricas, permitiendo una comprensión profunda y multifacética del tema abordado. La formación recibida ha sido integral, abarcando desde conceptos fundamentales hasta aplicaciones prácticas, lo que ha facilitado la identificación y resolución de problemas complejos, así como la implementación de metodologías adecuadas para la investigación.

Adicionalmente, la experiencia adquirida a lo largo de las prácticas de empresa durante el transcurso de la carrera ha sido un gran impulsor del mismo. Durante este periodo, se exploraron diversas facetas del ámbito profesional, consolidando conocimientos teóricos y habilidades prácticas en el área de las tecnologías de la información y la IA. En consonancia con esta formación práctica, surgió la semilla de la idea de crear un sistema para la implementación eficiente de redes neuronales, en respuesta a la creciente demanda de herramientas especializadas en este campo emergente.

Posteriormente, se llevó a cabo un periodo extraordinario de prácticas durante el verano en el *European Institute for Energy Research* (EIFER), donde se tuvo la oportunidad de aplicar los conocimientos adquiridos a lo largo de la carrera en un entorno profesional altamente especializado. Durante esta experiencia, se desarrolló e implementó una red neuronal, lo que permitió un acercamiento directo con un caso real y proporcionó una valiosa experiencia práctica que sirvió como fundamento para el proyecto que se presenta.

6.2. Metodología

El desarrollo de software es una actividad inherentemente compleja, caracterizada por requisitos variables, la necesidad de competencias especializadas y diversas, tecnología en constante evolución y la dificultad en la gestión de los equipos que enfrentan estos desafíos diariamente. Los procesos de desarrollo de sistemas se describen como complejos, impredecibles y mal definidos, lo que implica que no tienen entradas y salidas bien establecidas y, por tanto, son irrepetibles. En consecuencia, es crucial definir una metodología que guíe su desarrollo.

6.2.1. Agile

Agile es un tipo de metodología que utiliza el control empírico del proceso y considera el desarrollo de sistemas como sistemas adaptativos complejos. El control empírico de procesos es adecuado para entornos mal definidos, impredecibles e irrepetibles, implementando el control mediante inspecciones y adaptaciones frecuentes. La metodología ágil, como se presenta en la figura 6.1, se caracteriza por ciclos iterativos cortos, denominados *sprints*, que permiten una constante inspección y adaptación. Esta metodología es especialmente adecuada para proyectos de desarrollo de software debido a su capacidad para manejar cambios en los requisitos a través de la retroalimentación continua de las partes interesadas y la integración continua del código. La flexibilidad y rapidez proporcionadas por los ciclos iterativos permiten responder de manera efectiva a la evolución de las necesidades y a los cambios tecnológicos. Esta metodología se adapta a la perfección a los retos inherentes presentes en el ámbito de las redes neuronales donde se realizan actualizaciones constantes y es necesario adaptar los modelos a las mismas.

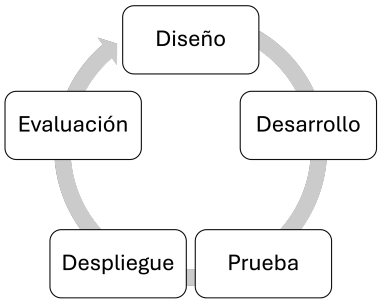

Figura 6.1: Etapas de la metodología ágil.

De esta forma, la naturaleza iterativa y adaptable de las metodologías ágiles ha permitido gestionar eficazmente la complejidad y la incertidumbre de *Flogo*, facilitando la entrega de incrementos funcionales del producto en plazos cortos. Además, se ha fomentado la colaboración activa entre el tutor y los otros miembros que desarrollaban el sistema, lo que ha enriquecido el proceso de desarrollo con diferentes perspectivas y conocimientos especializados. La colaboración ha proporcionado diferentes perspectivas y habilidades complementarias, lo que ha mejorado el proceso de desarrollo y ha contribuido a la calidad y robustez del sistema

final. Además, el trabajo en equipo ha fomentado un ambiente de apoyo mutuo y motivación, lo que ha sido fundamental para mantener el impulso y la dedicación a lo largo de todo el proyecto.

6.2.2. Gitflow

GitFlow es una metodología de ramificación que establece una serie de reglas y convenciones para gestionar las ramas en un repositorio *Git*, lo que facilita la colaboración y la integración continua en proyectos de software. En la figura 6.2 se observa que la metodología se basa en dos tipos principales de ramas: la rama *master*, que contiene el código de producción siempre estable y listo para ser desplegado, y la rama *develop* que es la rama de integración donde se combinan todas las nuevas características antes de ser lanzadas.

Además de las ramas principales, *Gitflow* utiliza ramas de soporte para gestionar diferentes aspectos del desarrollo. Un ejemplo son las ramas *feature*, las cuales se crean a partir de *develop* para desarrollar nuevas características y se fusionan de nuevo en *develop* cuando están completas. Otro tipo de ramas son las de *release*, las cuales se crean a partir de *develop* para preparar una nueva versión. Este enfoque estructurado y bien definido en la gestión de ramas garantiza una gestión eficiente del ciclo de vida del software, facilitando la colaboración entre los miembros del equipo y asegurando la estabilidad y la calidad del producto final.

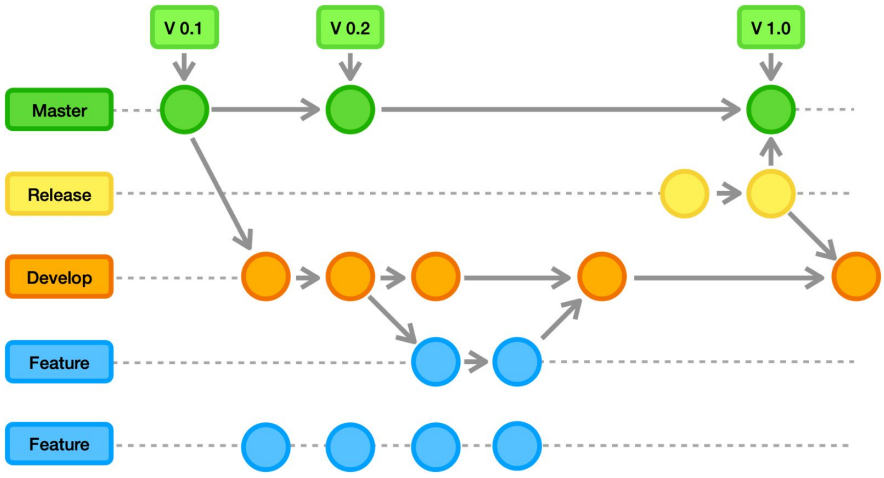

Figura 6.2: Diagrama de ramas de *Gitflow*.

El flujo de trabajo en *Gitflow* comienza con la creación de una rama *feature* para desarrollar una nueva funcionalidad. Una vez completada, esta rama se fusiona en *develop*. Cuando se ha alcanzado una cierta estabilidad en *develop* y se está listo para una nueva versión, se crea una *release* branch para preparar el lanzamiento final, hacer pruebas y corregir fallos menores. Al finalizar, se fusiona dicha rama en *master* y se etiqueta la versión correspondiente.

6.2.3. Semantic versioning

El versionado semántico es un sistema de numeración para versiones de software diseñado para comunicar de manera clara y precisa las modificaciones realizadas en el software a los usuarios y desarrolladores. Este sistema sigue un formato específico de tres números separados por puntos, representados como *MAJOR.MINOR.PATCH*, donde cada uno de estos números tiene un significado particular.

El primer número, *MAJOR*, se incrementa cuando se introducen cambios incompatibles con versiones anteriores, es decir, cuando las modificaciones realizadas pueden romper la compatibilidad del software con versiones previas. Un incremento en este número indica a los usuarios que deben esperar cambios significativos que podrían requerir ajustes en su código para funcionar con la nueva versión.

El segundo número, *MINOR*, se incrementa cuando se agregan nuevas funcionalidades de manera retrocompatible, lo que significa que no se afecta la compatibilidad con versiones anteriores. Los cambios menores indican la adición de nuevas características y mejoras que no deberían afectar el funcionamiento del software existente.

El tercer número, *PATCH*, se incrementa cuando se realizan correcciones de errores o pequeñas mejoras que no alteran las funcionalidades ni la compatibilidad del software. Los cambios de parche típicamente consisten en correcciones de errores o ajustes menores que no afectan el uso del software de manera significativa.

6.3. Cronología

La figura 6.3 presenta una línea de tiempo que abarca las fases y tareas del desarrollo del trabajo de fin de grado. Al proporcionar un marco temporal coherente, esta sección sirve como una herramienta esencial para contextualizar y analizar de manera más profunda las iteraciones realizadas a lo largo del proyecto.

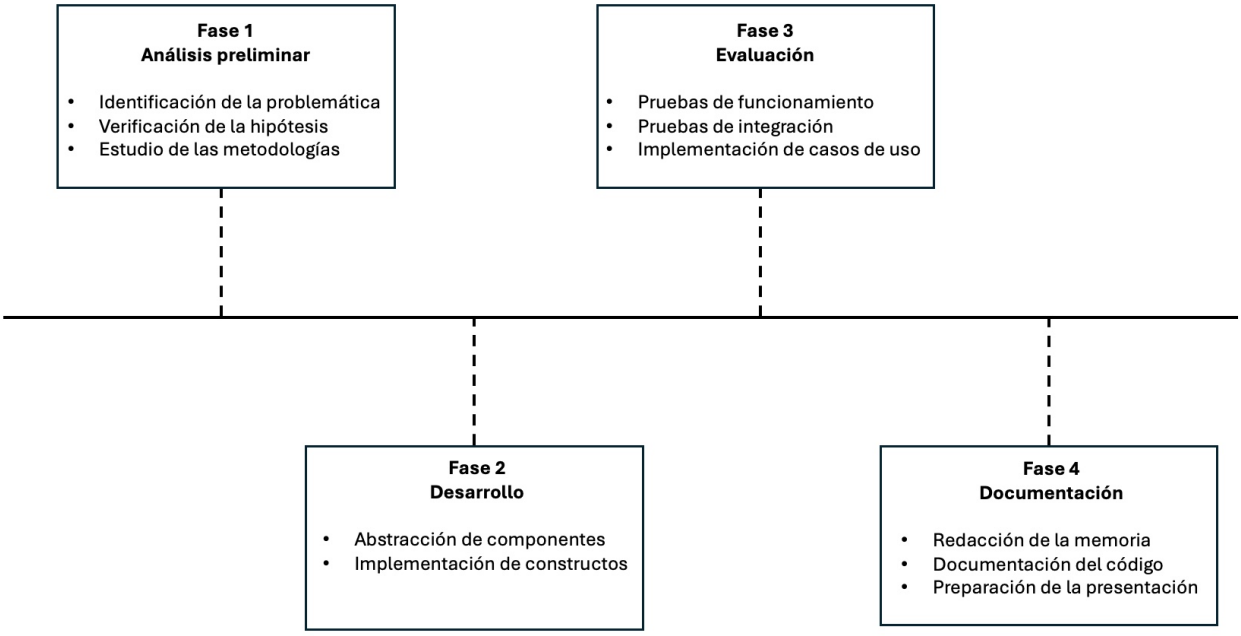

Figura 6.3: Línea temporal de las fases realizadas con sus respectivas tareas.

La primera fase del proyecto se centró en el estudio previo y análisis, que abarcó las dos primeras semanas del desarrollo aproximadamente. Durante este período, se identificó la problemática experimentada durante el estudio del grado relacionada con el hecho de que el proceso de diseño y entrenamiento de redes neuronales podría ser mucho más automatizable en comparación con su estado actual. A partir de esta observación, el siguiente paso fue verificar esta teoría sobre el escaso uso de la ingeniería del software en el ámbito de la IA. Para ello, se llevó a cabo un estudio sobre las principales metodologías utilizadas por la comunidad científica en la creación de sus propios modelos. Este análisis permitió reafirmar la hipótesis inicial confirmando el hecho de que no existe un estándar claro en estos procedimientos sino que cada programador se adapta al conjunto de herramientas que ha decidido usar, omitiendo muchos principios básicos, como son la repetición de código, lo que permitiría generar un código mucho más entendible y escalable.

En la segunda fase, abarcando las siguientes nueve semanas, se procedió al diseño, desarrollo e implementación del proyecto. Como resultado de la primera fase, se acordó desarrollar un sistema que encapsularía el uso de los entornos de trabajo habituales en DL y permitiría la automatización de prácticas típicas que suelen llevar a errores. La comunicación del usuario final con este sistema se realizaría a través de un DSL, lo cual reduciría la cantidad de conocimiento necesario por parte del programador para crear su propia red neuronal. Este estaría apoyado en un *framework* que permitiría ejecutar las definiciones realizadas en el mismo. Durante esta fase, surgió la necesidad de dividir dicho *framework* en dos partes debido a su gran complejidad: una para el diseño de las redes neuronales y otra para la gestión de su ciclo de vida. Una vez dividido el sistema en partes, se dividió el trabajo entre los diferentes integrantes del grupo para su desarrollo. En el caso de este trabajo de fin de grado, se comenzó por la abstracción de los componentes de diferentes arquitecturas de redes neuronales mediante el

estudio de una amplia variedad de tipos diferentes. Estas tareas se desarrollaron de manera iterativa, permitiendo revisiones y ajustes basados en los hallazgos obtenidos durante cada *sprint*. De esta forma, se empezó con la abstracción de las ANN y la implementación de los constructos asociados con dicho tipo de arquitectura, para continuar con los relacionados con la CNN y acabar con los de la RNN. Esta etapa incluyó la elaboración de la arquitectura del *framework*, la selección de tecnologías y herramientas a utilizar, y la creación de prototipos iniciales que permitieran visualizar y validar las ideas conceptuales.

Durante todo el proceso de desarrollo, las reuniones periódicas entre los integrantes del grupo y el tutor resultaron fundamentales. Estas reuniones permitieron acordar los constructos que se iban a utilizar para cada una de las partes del sistema, asegurando así la correcta integración y conexión de todas las partes desarrolladas. La colaboración y comunicación continua con el tutor facilitó la identificación de posibles problemas y la implementación de soluciones efectivas, lo cual fue crucial para mantener la coherencia y funcionalidad del sistema en su conjunto. Además, estas reuniones fomentaron un entorno de trabajo colaborativo y una mayor alineación entre los miembros del equipo, contribuyendo al éxito del proyecto.

La tercera fase, desarrollada durante las siguientes dos semanas, se enfocó en la evaluación, validación y prueba del sistema. Durante este periodo, se realizaron iteraciones específicas para la realización de un conjunto de pruebas exhaustivas que corroboraran el correcto funcionamiento del sistema. Se diseñaron y ejecutaron pruebas de funcionamiento, de integración y de sistema para asegurar que todos los componentes operaban de manera conjunta sin errores. También se crearon varios casos de uso posibles, abarcando escenarios típicos y atípicos, lo que permitió una evaluación exhaustiva y minuciosa del rendimiento y la robustez del proyecto. La implementación de estas pruebas no solo garantizó la calidad del software, sino que también facilitó la detección y corrección de errores antes de la fase final de entrega.

Finalmente, en la cuarta fase, correspondiente a la documentación y presentación, que ocupó alrededor de una semana, se procedió con la redacción de la memoria del trabajo de fin de grado y la documentación del código en el repositorio. Se elaboró una documentación detallada que incluyó tanto aspectos técnicos como funcionales del *framework*, asegurando que cualquier futuro usuario o desarrollador pudiera comprender y utilizarlo de manera efectiva. Además, se preparó la presentación para su futura exposición delante del tribunal, incluyendo una síntesis de los objetivos, la metodología, los resultados y las conclusiones del proyecto. Esta etapa culminó con la revisión final del trabajo y la confirmación de que todos los requisitos y expectativas del proyecto habían sido satisfechos.

6.4. Herramientas

El marco tecnológico del trabajo de fin de grado proporciona el contexto y la infraestructura necesarios para el desarrollo y la implementación del *framework* desarrollado. En este marco, se definen las herramientas, tecnologías y recursos utilizados para construir el sistema propuesto, garantizando su viabilidad y eficacia.

6.4.1. Github

En el desarrollo del trabajo de fin de grado, se ha hecho uso de *Github* como plataforma central para la gestión del código fuente y la colaboración entre los miembros del equipo. *Github* es un sistema de control de versiones distribuido que permite a los desarrolladores almacenar, compartir y colaborar en proyectos de software de manera eficiente y efectiva. En su núcleo, *Github* se basa en el concepto de repositorios, que son almacenes de código donde se almacenan todas las versiones de los archivos y se registra el historial de cambios a lo largo del tiempo.

En cuanto al sistema de control de versiones, esta es una herramienta que registra los cambios realizados en archivos a lo largo del tiempo. Esto permite a los usuarios rastrear las modificaciones, revertir a versiones anteriores, comparar cambios entre versiones y colaborar de manera eficiente en proyectos de software u otros tipos de documentos.

Una de las características clave de *Github* es su capacidad para facilitar el trabajo colaborativo mediante el uso de ramas, que permiten a los desarrolladores trabajar en diferentes aspectos del proyecto de forma paralela sin interferir con el trabajo de los demás.

6.4.2. Python

Python ha sido elegido como el lenguaje de programación principal para el desarrollo del *framework* por varias razones fundamentales. En primer lugar, *Python* es conocido por su sintaxis clara y legible, lo que lo convierte en un lenguaje fácil de entender y aprender, especialmente para aquellos que están comenzando en el campo del desarrollo de software. Esta claridad en la escritura del código facilita la colaboración entre los miembros del equipo y la comprensión del código desarrollado.

Otra razón importante para elegir *Python* es su amplia adopción en la comunidad de IA y ML. *Python* se ha convertido en el lenguaje de facto en estos campos debido a su flexibilidad, potencia y facilidad de uso. Esta popularidad se traduce en una abundancia de recursos, documentación y comunidad de desarrolladores que facilitan el aprendizaje, la resolución de problemas y la colaboración en proyectos relacionados con las redes neuronales.

6.4.3. Pytorch

PyTorch es la librería de ML usada por el *framework*, siendo la misma de código abierto y habiéndose convertido en una de las herramientas más populares para el desarrollo de aplicaciones de IA y DL debido a su potencia y capacidad para ejecutar cómputos en tarjetas gráficas. Una de las principales razones que condujeron a la elección de *Pytorch* sobre otras librerías fue su capacidad significativa de personalización en el proceso de creación y entrenamiento de modelos. En contraste con alternativas disponibles, *Pytorch* posibilita una programación a un nivel más bajo, lo que proporciona una mayor flexibilidad para adaptarse a preferencias y necesidades específicas. Esta flexibilidad ha permitido ajustar y optimizar los

modelos con mayor precisión, aprovechando al máximo las capacidades del *framework* para los objetivos de desarrollo.

6.4.4. Pycharm

Pycharm [8] es un IDE de *Python* comercializado por *JetBrains*, disponible en una versión bajo la licencia Apache 2 y otra de pago. *Pycharm* es una herramienta altamente especializada y potente diseñada específicamente para el desarrollo en *Python*. Su elección se basa en una serie de ventajas significativas que ofrece para el desarrollo de proyectos en el mismo.

En primer lugar, *Pycharm* proporciona un conjunto completo de características que facilitan la escritura, edición y depuración de código *Python*. Su interfaz intuitiva y amigable ofrece funcionalidades avanzadas, como resaltado de sintaxis, completado automático de código, depuración interactiva y análisis estático de código, que aumentan la productividad del desarrollador y ayudan a evitar errores comunes.

Otra ventaja de *Pycharm* es su integración con sistemas de control de versiones como *Git*, lo que facilita la colaboración en equipo y la gestión del código fuente. Su capacidad para trabajar con repositorios remotos y gestionar ramificaciones y fusiones de forma eficiente simplifica el flujo de trabajo colaborativo en proyectos de desarrollo de software.

6.4.5. ItRules

ItRules [10] es el motor de plantillas usado para la generación de codigo, el cual está basado en *Java* y desarrollado en el Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (SIANI) que puede generar texto de cualquier tipo a partir de plantillas. De esta forma, el código *Java* puede separarse de la vista, permitiendo desarrollar aplicaciones según el *Model View Controller* (MVC).

Capítulo 7

Conclusiones y trabajo futuro

Este capítulo ofrece una síntesis de los hallazgos y resultados obtenidos, así como una visión prospectiva sobre posibles direcciones futuras de investigación. En esta sección, se presenta un análisis objetivo de los resultados alcanzados en relación con los objetivos planteados inicialmente, proporcionando una evaluación crítica de su relevancia y contribución al campo de estudio en cuestión. Además, se plantean líneas de trabajo futuro que permitirían ampliar y profundizar en la comprensión del problema abordado, identificando áreas de investigación emergentes y posibles mejoras en la metodología empleada. Este análisis riguroso y prospectivo es fundamental para consolidar las conclusiones del estudio y orientar el desarrollo de investigaciones futuras en esta área específica.

En el desarrollo del presente trabajo se ha implementado con éxito el sistema objetivo, *Flogo*, al unir los diferentes componentes, obteniendo de esta forma un DSL que permite definir de manera eficiente tanto la estructura como el ciclo de vida de las redes neuronales para poder obtener modelos entrenados. El DSL desarrollado ofrece funcionalidades avanzadas que permiten a los usuarios especificar las distintas capas y parámetros de una red neuronal. Además, incorpora mecanismos para gestionar las diferentes fases del ciclo de vida de las redes, incluyendo la definición, el entrenamiento, la evaluación y la mejora continua de los modelos. Posteriormente, los *frameworks* estructural y operacional permiten que las definiciones realizadas se puedan llevar a cabo con éxito.

7.1. Resultados

Si se observa la figura 7.1 de una red implementada en *Pytorch* en comparación con la figura 7.2 de la misma red una red implementada con el *framework* se puede apreciar a simple vista el hecho de que el desarrollo del *framework* ha permitido lograr significativos avances en términos de legibilidad. Esto sumado a la capacidad de encapsulación tecnológica conseguida gracias a su diseño modular y a una estructura de código bien organizada, permitirá a los desarrolladores comprender y mantener el código con mayor facilidad. Además, este *framework* actúa como una capa de abstracción que aísla las complejidades de la tecnología subyacente,

permitiendo a los equipos enfocarse en la lógica de negocio sin preocuparse por los detalles técnicos específicos. Esto no solo optimiza el proceso de desarrollo, sino que también facilita la integración y adaptación a futuras tecnologías.

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=33, kernel_size=(3, 3), stride=(2, 2))
        self.relu1 = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(kernel_size=(5, 5), stride=(4, 4))
        self.conv2 = nn.Conv2d(in_channels=33, out_channels=16, kernel_size=(3, 3), stride=(3, 3))
        self.relu2 = nn.ReLU()
        self.maxpool2 = nn.MaxPool2d(kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        self.flatten = nn.Flatten(start_dim=1, end_dim=-1)
        self.linear = nn.Linear(in_features=64, out_features=2)
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.batchnorm1(x)
        x = self.maxpool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.batchnorm2(x)
        x = self.maxpool2(x)
        x = self.flatten(x)
        x = self.linear(x)
        x = self.softmax(x)
        return x
```

Figura 7.1: Implementación de una red neuronal convolucional en *Pytorch*.

```
(Architecture("CNN")
     .attach(ConvolutionalSection([
        Block([
           ConvolutionalLayer(in_channels=3, out_channels=33, kernel=(3, 3), stride=(2, 2), padding=(0, 0)),
           ReLULayer(),
           BatchNormalizationLayer(num_features=33, eps=1.0E-5, momentum=0.1),
           MaxPoolLayer(kernel=(5, 5), stride=(4, 4), padding=(0, 0))
        ]),
        Block([
           ConvolutionalLayer(in_channels=33, out_channels=16, kernel=(3, 3), stride=(3, 3), padding=(0, 0)),
           ReLULayer(),
           BatchNormalizationLayer(num_features=16, eps=1.0E-5, momentum=0.1),
           MaxPoolLayer(kernel=(3, 3), stride=(2, 2), padding=(1, 1))
        ])
      ]))
      .attach(FlattenLayer(from_dim=3, to_dim=1))
      .attach(LinearSection([
        Block([
           LinearLayer(in_features=64, out_features=2, dimension=-1, bias=True),
           SoftmaxLayer(dimension=-1)
        ])
      ])))
```

Figura 7.2: Implementación de una red neuronal convolucional en el *framework*.

7.1.1. Objetivos cumplidos

Es satisfactorio afirmar que se han alcanzado con éxito los objetivos iniciales planteados para este proyecto como se puede observar en la tabla 7.1. El desarrollo del sistema ha resultado en una herramienta que proporciona a los desarrolladores un mayor control y autonomía sobre el diseño de sus redes neuronales, cumpliendo así con el primer objetivo establecido. Este logro no solo mejora la capacidad de respuesta a los cambios del mercado, sino que también fomenta la innovación y la experimentación en el campo de la inteligencia artificial al liberar a los desarrolladores de las restricciones impuestas por las soluciones actuales.

Cuadro 7.1: Relación de los objetivos propuestos con su cumplimiento a lo largo del trabajo de fin de grado.

Código	Objetivo	Realización
O1	Mayor control y autonomía	✓
O2	Flexibilidad y adaptabilidad	✓
O3	Independencia tecnológica	✓
O4	Facilidad de uso y personalización	✓

En cuanto a la flexibilidad y adaptabilidad, el *framework* ha demostrado ser capaz de proporcionar una plataforma que permite a los usuarios implementar y modificar estructuras de red con facilidad, adaptándose a las nuevas tendencias y tecnologías sin depender de actualizaciones de terceros, tal como se ha establecido en los primeros tres objetivos. Esta característica es fundamental para garantizar que los desarrolladores puedan mantenerse al día con los avances en el campo del ML sin estar limitados por las restricciones de las librerías comerciales.

Por último, el *framework* ha logrado cumplir con el objetivo de ofrecer una interfaz intuitiva y opciones de personalización que permiten a los desarrolladores diseñar redes neuronales adaptadas a sus necesidades específicas. Esta facilidad de uso y personalización es esencial para garantizar que los usuarios puedan aprovechar al máximo las capacidades del mismo y desarrollar modelos de DL que se ajusten a sus requisitos individuales, sin verse limitados por las restricciones de las librerías comerciales disponibles en el mercado. En conjunto, el cumplimiento de estos objetivos demuestra el éxito del proyecto en proporcionar una solución que aborda las necesidades y desafíos actuales en el diseño y desarrollo de redes neuronales.

7.1.2. Contribuciones al ámbito

El trabajo desarrollado presenta una serie de contribuciones significativas al ámbito de la IA y, en particular, al diseño y manejo de redes neuronales. En primer lugar, se ha logrado que el código resultante sea notablemente más legible. El DSL implementado se distingue por su simplicidad en comparación con cualquier lenguaje de propósito general, lo cual contribuye a una curva de aprendizaje más suave para los desarrolladores. Esta facilidad de uso permite que incluso aquellos con menos experiencia en programación puedan entender y utilizar el sistema con eficiencia.

Otra contribución destacable es la estandarización del proceso de creación y entrenamiento de redes neuronales. *Flogo* proporciona un marco uniforme que guía a los usuarios a través de cada fase del ciclo de vida de una red neuronal, desde su definición inicial hasta su mejora continua. Esta estandarización no solo facilita la comunicación entre los equipos de trabajo, eliminando cualquier brecha semántica, sino que también asegura la consistencia y la coherencia en los proyectos desarrollados.

Además, el sistema permite explorar diferentes arquitecturas de redes neuronales de manera más rápida y efectiva mediante la implementación de laboratorios y experimentos. Esta capacidad de experimentar de forma ágil es crucial para la innovación y el avance en el campo, permitiendo la prueba y evaluación de múltiples configuraciones con un esfuerzo mínimo.

La reducción de costes de mantenimiento es otra de las contribuciones significativas. La automatización de procesos repetitivos y la disminución de las exigencias técnicas necesarias para gestionar redes neuronales permiten que los recursos se utilicen de manera más eficiente. Al mismo tiempo, *Flogo* aísla a los usuarios de las complejidades tecnológicas subyacentes, aumentando la mantenibilidad de los sistemas desarrollados. Esta abstracción tecnológica no solo facilita el mantenimiento, sino que también contribuye a la sostenibilidad a largo plazo de las soluciones implementadas.

7.1.3. Lecciones aprendidas

A lo largo del desarrollo de este proyecto, se ha adquirido una valiosa experiencia en el trabajo en grupo, fortaleciendo la capacidad para colaborar de manera efectiva en un entorno multidisciplinario. La coordinación con los compañeros de equipo y la integración de diversas perspectivas y habilidades han sido fundamentales para el éxito del proyecto, promoviendo un ambiente de cooperación y aprendizaje mutuo.

La realización de este proyecto, de considerable envergadura y complejidad, ha demostrado la capacidad para llevar a cabo y gestionar iniciativas de gran alcance. La planificación, ejecución y supervisión de un proyecto de tal magnitud han permitido desarrollar habilidades de liderazgo y gestión de proyectos, asegurando el cumplimiento de los objetivos establecidos y la entrega de resultados de alta calidad.

Asimismo, este trabajo ha requerido una notable capacidad de abstracción para abordar problemas complejos y diseñar soluciones eficientes. La implementación del lenguaje ha exigido la conceptualización de modelos abstractos y su traducción a un lenguaje accesible y práctico, mejorando así la capacidad para resolver problemas técnicos mediante enfoques innovadores.

Finalmente, la experiencia obtenida ha despertado un interés por la investigación. El proceso de exploración, experimentación y análisis realizado durante el proyecto ha fomentado una actitud inicial y un deseo de profundizar en el conocimiento, motivando la aspiración de emprender futuros trabajos de investigación. Este interés se ha visto reflejado en la dedicación y el entusiasmo puestos en la consecución de los objetivos del proyecto, y en la búsqueda continua de nuevos desafíos y oportunidades de aprendizaje en el campo de la IA

y el desarrollo de redes neuronales.

7.2. Extensiones

Los objetivos futuros están orientados hacia la expansión del sistema. En primer lugar, aumentar la topología de redes posibles implementables en el sistema, con el propósito de explorar y desarrollar arquitecturas que puedan abordar problemas más complejos y específicos. Asimismo, se planea incorporar métodos avanzados de aprendizaje, como el *transfer learning* y el *fine tuning*. La implementación del *transfer learning* facilitará la reutilización de conocimientos adquiridos en una tarea para mejorar el aprendizaje en otra tarea relacionada a la inicial, optimizando así el proceso de entrenamiento y aumentando la capacidad de generalización de los modelos. Por otro lado, la implementación del *fine tuning* permitirá ajustar modelos preentrenados a nuevos problemas con un menor requerimiento de datos, lo cual resulta especialmente útil en dominios donde los conjuntos de datos son pequeños o difíciles de recopilar.

Otro aspecto fundamental será el avance en el desarrollo de arquitecturas parametrizadas. Se continuará trabajando en la creación y refinamiento de modelos que puedan modificar su estructura durante el entrenamiento basándose en algoritmos genéticos, intentando introducir, a su vez, mecanismos automatizados que operen dentro del espacio de parámetros del modelo con el objetivo de optimizarlos. Estos mecanismos buscarán activamente los mejores parámetros durante el proceso de entrenamiento. El objetivo es automatizar la búsqueda y ajuste de hiperparámetros para maximizar el rendimiento del modelo sin necesidad de una intervención manual extensa.

Por otro lado, se contempla la opción de explorar la implementación de otras librerías de redes neuronales tanto si son existentes como de futura creación, incluso desarrolladas desde cero, con el objetivo de ofrecer a los usuarios una mayor personalización y flexibilidad. Esto implicaría la integración de librerías que presenten diferentes enfoques de diseño, optimización y funcionalidades específicas, lo que permitiría a los usuarios seleccionar la librería más adecuada para sus necesidades y preferencias. La incorporación de estas nuevas librerías ampliaría el abanico de opciones disponibles para los usuarios, brindándoles la posibilidad de aprovechar al máximo las características y capacidades únicas de cada una.

Finalmente, se contempla la promoción de la capacitación en el uso de estas nuevas herramientas y técnicas a través de cursos de formación tanto físicos como online. Estos cursos estarán diseñados para diferentes niveles de experiencia, desde principiantes hasta expertos, con el objetivo de crear una comunidad bien informada y competente en el uso de tecnologías avanzadas de redes neuronales y ML. De esta manera, se espera no solo avanzar técnicamente, sino también fomentar el conocimiento y la adopción de estas innovaciones en un ámbito más amplio.

Bibliografía

[1] 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.

[2] Bebis, G. and Georgiopoulos, M. (1994). Feed-forward neural networks. *Ieee Potentials*, 13(4):27–31.

[3] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[4] de Las Palmas de Gran Canaria, U. (2020). *Grado en Ciencia e Ingeniería de Datos: Memoria del Plan de Estudios*. Escuela de Ingeniería Informática, Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, España.

[5] Dubey, S. R., Singh, S. K., and Chaudhuri, B. B. (2022). Activation functions in deep learning: A comprehensive survey and benchmark. *Neurocomputing*, 503:92–108.

[6] Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

[7] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[8] JetBrains (2024). Pycharm. Consultado el 31 de mayo de 2024.

[9] Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier.

[10] José Juan Hernández Cabrera, José Évora Gómez, O. R. A. and Ramirez, M. C. (2015). Itrules. https://bitbucket.org/siani/itrules-java. Consultado el 30 de mayo de 2024.

[11] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

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

[13] Lieberherr, K., Holland, I., and Riel, A. (1988). Object-oriented programming: An objective sense of style. *ACM Sigplan Notices*, 23(11):323–334.

[14] Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, 1(34):597.

[15] Naveed, H., Arora, C., Khalajzadeh, H., Grundy, J., and Haggag, O. (2024). Model driven engineering for machine learning components: A systematic literature review. *Information and Software Technology*, page 107423.

[16] 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.

[17] Santana, J. C. S. (2024). Framework. https://github.com/NeuralFlogo/framework. Consultado el 10 de junio de 2024.

[18] Schmidt, D. C. et al. (2006). Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25.

Glosario

Camel Case Práctica de escribir frases sin espacios ni signos de puntuación y con cada palabra en mayúscula. 30

Deep Learning Rama del aprendizaje automático que emplea redes neuronales multicapa, conocidas como redes neuronales profundas, con el objetivo de aprender representaciones complejas. 1

Domain-Specific Language Lenguaje de programación con un mayor nivel de abstracción optimizado para una clase específica de problema que utiliza los conceptos y reglas del campo o dominio para el que se implementan. 2

fine tuning Técnica de ML donde un modelo preentrenado en una tarea general se adapta o se entrena adicionalmente para una tarea específica. 53

Fluent Interface API orientada a objetos cuyo diseño se basa en gran medida en el encadenamiento de métodos para aumentar la legibilidad del código.. 24

framework Abstracción en la que el software, que proporciona funcionalidad genérica, puede modificarse selectivamente mediante código adicional escrito por el usuario, proporcionando así software específico para la aplicación. 2

Git Sistema distribuido de control de versiones que rastrea las versiones de los archivos. 43

IDE Aplicación de software que proporciona facilidades completas para el desarrollo de software en uno o varios lenguajes de programación. 48

Machine Learning Disciplina de la inteligencia artificial y la informática que se enfoca en utilizar datos y algoritmos para que la inteligencia artificial reproduzca el modo en que los humanos aprenden, incrementando progresivamente su precisión. 1

Model Driven Engineering Metodología de desarrollo de software que se centra en la creación y explotación de modelos de dominio, que son modelos conceptuales de todos los temas relacionados con un problema específico. 8

Model View Controller Patrón de diseño software que separa una aplicación en tres componentes interconectados y pone énfasis en la separación entre la lógica de negocio del software y la presentación visual. 48

SOLID Acrónimo que designa cinco principios de diseño destinados a hacer más comprensibles, flexibles y mantenibles los diseños orientados a objetos. 11

transfer learning Técnica de ML en la cual un modelo desarrollado para una tarea específica se reutiliza como punto de partida para un modelo en una segunda tarea relacionada. Este enfoque aprovecha el conocimiento adquirido en la tarea inicial para mejorar el rendimiento o reducir el tiempo de entrenamiento en la nueva tarea. 53

UML Lenguaje de modelado visual de uso general que pretende ofrecer una forma estándar de visualizar el diseño de un sistema.. VII, 14

Apéndices

Apéndice A

Capas de activación

A.1. Capa sigmoide

La función sigmoide [5] es una función matemática continua que tiene una forma característica de "S". Se define comúnmente como:

$$f(x) = \frac{1}{1 + e^{-x}}$$

donde "e" es la base del logaritmo natural y "x" es la variable independiente. La función sigmoide mapea cualquier número real "x" a un valor en el rango de 0 a 1. Esta función es ampliamente empleada debido a su aplicación frecuente en modelos donde se requiere predecir la salida como una probabilidad, dado que la probabilidad está limitada al rango de 0 a 1. Además, esta función es diferenciable y produce un gradiente suave, lo que significa que evita bruscos cambios en los valores de salida, como se observa en su forma de "S".

A.2. Capa tangente hiperbólica

La función tangente hiperbólica [5], denotada comúnmente como tanh(x), es una función matemática que se define como el cociente entre el seno hiperbólico y el coseno hiperbólico de un número real x.

$$f(x) = \frac{sinh(x)}{cosh(x)}$$

La función de tangente hiperbólica tiene propiedades análogas a la tangente trigonométrica, como simetría impar, límites asintóticos, y derivadas relacionadas. Es una función continua y su rango es el intervalo (-1, 1). Suele usarse en capas ocultas de una red neuronal ya que sus valores se encuentran entre -1 y 1, por lo tanto, la media de la capa oculta resulta ser 0 o

muy cercana lo que ayuda a centrar los datos y facilita mucho el aprendizaje para la siguiente capa.

A.3. Capa ReLU

La función de activación *Rectified Linear Unit* (ReLU) [5], aunque da la impresión de una función lineal, ReLU tiene una función derivada y permite la retropropagación al mismo tiempo que la hace computacionalmente eficiente. Formalmente se define como:

$$f(x) = max(0, x)$$

donde "x" es la entrada y se realiza el máximo entre 0 y dicha entrada. La función ReLU no activa todas las neuronas al mismo tiempo, sino que produce una salida igual a cero cuando la entrada es negativa y conserva la entrada original cuando esta es positiva. Esto significa que solo activa neuronas cuando la entrada es mayor que cero, lo que facilita el aprendizaje y mejora la eficiencia computacional, ya que es computacionalmente más económica que otras funciones de activación como la función sigmoide o la tangente hiperbólica.

La función ReLU tiene varias propiedades deseables, como la simplicidad computacional, la convergencia rápida durante el entrenamiento y la capacidad de resolver el problema de desvanecimiento del gradiente. Sin embargo, puede sufrir del problema conocido como "neuronas muertas" en el que ciertas neuronas pueden quedar inactivas si su salida permanece en cero durante un largo período de tiempo.

A.4. Capa leaky ReLU

La función de activación *Leaky Rectified Linear Unit* (*Leaky* ReLU) [5] es una variante de la función ReLU que busca abordar el problema de "neuronas muertas" que puede surgir en la función ReLU. Formalmente, se define como:

$$f(x) = \begin{cases} x & \text{si } x \geq 0, \\ \alpha \cdot x & \text{si } x < 0, \end{cases}$$

donde "x" es la entrada y "α" es un parámetro pequeño y positivo que se utiliza para definir la pendiente de la función en el tramo negativo. La función *Leaky* ReLU activa neuronas cuando la entrada es mayor que cero, pero en lugar de hacer que la salida sea cero cuando la entrada es negativa, permite un valor pequeño y proporcional a la entrada negativa, controlado por el parámetro "α". Esto permite un flujo de gradiente incluso para las entradas negativas, lo que puede ayudar a mitigar el problema de las "neuronas muertas" y mejorar la capacidad de aprendizaje de la red neuronal.

A.5. Capa ELU

La función de activación *Exponential Linear Unit* (ELU) [5] es otra alternativa a la función ReLU, diseñada para abordar algunas de sus limitaciones, como el problema de "neuronas muertas". Formalmente, se define como:

$$f(x) = \begin{cases} x & \text{si } x \geq 0, \\ \alpha \cdot (exp(x) - 1) & \text{si } x < 0, \end{cases}$$

donde "x" es la entrada y "α" es un parámetro positivo que determina la pendiente de la función en el tramo negativo. La función ELU activa neuronas cuando la entrada es mayor que cero, similar a ReLU. Sin embargo, en lugar de ser lineal en el tramo negativo, utiliza una función exponencial para suavizar la transición. Esto ayuda a evitar el problema de "neuronas muertas" y puede mejorar el rendimiento del modelo. Además, la función ELU es diferenciable en todos los puntos, lo que facilita el entrenamiento mediante métodos de optimización basados en gradiente.

A.6. Capa GELU

La función de activación *Gaussian Error Linear Unit* (GELU) [5] es una función de activación no lineal que se define formalmente como:

$$\text{GELU}(x) = 0{,}5 \cdot x \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0{,}044715 \cdot x^3)\right)\right)$$

donde "x" es la entrada. La función GELU activa las neuronas aplicando una transformación no lineal a la entrada. Se caracteriza por su suavidad y similitud con la función de activación sigmoide, pero evita la saturación en el extremo negativo de su dominio, lo que puede ayudar a evitar el problema del desvanecimiento del gradiente y mejorar el rendimiento del modelo durante el entrenamiento.

A.7. Capa SELU

La función de activación *Scaled Exponential Linear Unit* (SELU) [5] es una función de activación que se caracteriza por ser auto-normalizante, lo que significa que mantiene la salida de una capa dentro de una distribución estable, independientemente de la magnitud de las entradas. Formalmente, se define como:

$$\text{SELU}(x) = \lambda \begin{cases} x & \text{si } x > 0, \\ \alpha e^x - \alpha & \text{si } x \leq 0, \end{cases}$$

donde "x" es la entrada y "λ" y "α" son constantes predefinidas que ajustan la escala y la pendiente de la función, respectivamente. La función SELU es una versión escalada de la función ELU y tiene la propiedad única de mantener la salida de una capa cerca de una media cero y una varianza uno. Esto es esencial para garantizar la estabilidad del entrenamiento en redes neuronales profundas. Además, SELU tiene la capacidad de autonormalización, lo que significa que las estadísticas de las activaciones se mantienen estables a medida que se propagan a través de múltiples capas de la red.

A.8. Capa swish

La función de activación *Swish* [5] es una función no lineal propuesta recientemente. Formalmente, se define como:

$$\text{Swish}(x) = x \cdot \sigma(\beta x),$$

donde "x" es la entrada, "σ" representa la función sigmoidal y "β" es un parámetro que controla la pendiente de la función en el origen. La función Swish combina la no linealidad de la función sigmoidal con la simplicidad de la función identidad, lo que la hace fácil de computar y derivar. Al introducir la no linealidad a través de la función sigmoidal, *Swish* puede capturar relaciones más complejas entre las características de entrada, lo que puede mejorar el rendimiento del modelo.

Una característica interesante de *Swish* es que, a diferencia de algunas otras funciones de activación, no satura para valores muy grandes o muy pequeños de la entrada, lo que puede ayudar a evitar el problema del desvanecimiento del gradiente y promover un entrenamiento más estable en redes neuronales profundas.

A.9. Capa softmax

La función *Softmax* [5] es una función de activación utilizada para convertir un vector de números reales en una distribución de probabilidad. Formalmente, se define como:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}},$$

donde "x_i" es el valor de entrada en la posición i, "x_j" es el valor de entrada en la posición j, y "N" es el número total de elementos en el vector de entrada. La función *Softmax* toma

como entrada un vector de valores reales y produce como salida un vector de la misma dimensión, donde cada elemento está en el rango [0, 1] y la suma de todos los elementos es igual a 1, lo que lo convierte en una distribución de probabilidad válida.

La función *Softmax* es comúnmente utilizada en la capa de salida de una red neuronal cuando se aborda un problema de clasificación multiclase, donde cada neurona de salida representa la probabilidad de pertenecer a una clase específica. Al aplicar *Softmax*, se obtiene una distribución de probabilidad sobre todas las posibles clases, lo que facilita la interpretación y la toma de decisiones basada en las salidas de la red.

A.10. Capa GLU

La función de activación *Gated Linear Unit* (GLU) [5] es una función de activación que se define formalmente como:

$$\mathrm{GLU}(x) = \sigma(x) \otimes x,$$

donde se aplica la operación de multiplicación elemento a elemento a "x", que es la entrada y "σ", que es una función de activación sigmoide. La función GLU actúa como un mecanismo de puerta que permite que la información relevante fluya a través de la red mientras atenúa la información menos relevante. La función sigmoide controla la cantidad de información que se propaga, actuando como una puerta de activación que modula la entrada. Esta multiplicación punto a punto, por lo tanto, selecciona las partes de la entrada que son relevantes y las multiplica por el valor de activación correspondiente.

Apéndice B

Capas recurrentes

B.1. LSTM

La *Long Short-Term Memory* (LSTM) es un tipo especial de capa recurrente capaz de aprender dependencias a largo plazo [7]. Está compuestas por una estructura recurrente de unidades de memoria, cada una de las cuales contiene una celda de memoria y tres puertas: la puerta de entrada, la puerta de olvido y la puerta de salida.

Si se observa la figura B.1, la celda de memoria es el componente crucial de las LSTM y actúa como una "cinta transportadora" que puede mantener información durante largos períodos de tiempo. La LSTM tiene la capacidad de eliminar o añadir información a la celda de memoria, cuidadosamente regulada por estructuras llamadas puertas. Las puertas son una forma de dejar pasar información de forma opcional. Se componen de una capa de red neuronal sigmoide y una operación de multiplicación puntual. La capa sigmoide produce números entre cero y uno, que describen la cantidad de cada componente que debe dejarse pasar.

Una LSTM tiene tres de estas puertas, para proteger y controlar el estado de la memoria. La puerta de entrada controla cuánta nueva información se va a agregar a la celda de memoria en cada paso de tiempo. La puerta de olvido determina cuánta información existente en la celda de memoria se debe desechar. Y finalmente, la puerta de salida regula cuánta información de la celda de memoria se va a transmitir a la salida de la capa LSTM en cada paso de tiempo.

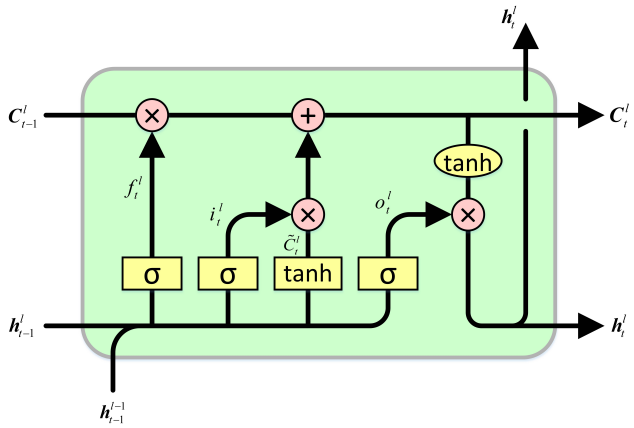

Figura B.1: Funcionamiento de una capa LSTM.

B.2. GRU

La *Gated Recurrent Unit* (GRU) es un tipo de capa recurrente que aborda el problema de la desaparición del gradiente mediante el uso de un mecanismo de puertas para controlar el flujo de información a través del tiempo [3]. Esta capa está diseñada para simplificar la arquitectura de las LSTM al combinar las puertas de entrada y olvido en una sola entidad llamada puerta de actualización, y fusionar la celda de memoria y la puerta de salida en una unidad única.

Como se observa en la figura B.2, cada unidad GRU contiene una puerta de actualización y una puerta de reinicio, que son controladas por funciones de activación sigmoidales. La puerta de actualización determina cuánta información de la celda de memoria anterior se va a combinar con la nueva entrada en cada paso de tiempo, mientras que la puerta de reinicio controla cuánta información de la entrada actual se va a tener en cuenta para actualizar la celda de memoria.

Este diseño simplificado permite a las capas GRU aprender representaciones complejas de secuencias de datos mientras reduce la cantidad de parámetros entrenables en comparación con las LSTM, lo que las hace más eficientes computacionalmente y más fáciles de entrenar en conjuntos de datos grandes. Las capas GRU son ampliamente utilizadas en aplicaciones de modelado de secuencias, como el procesamiento del lenguaje natural, la traducción automática y la generación de texto.

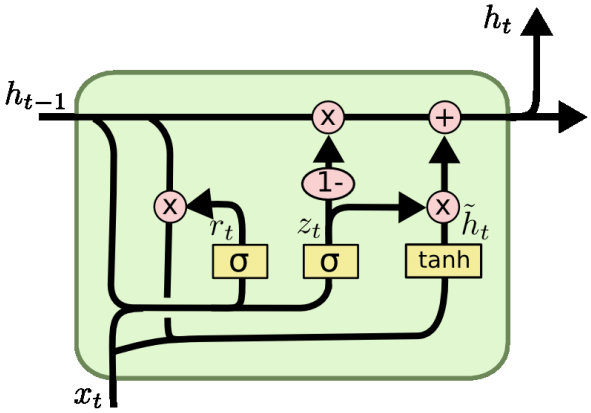

Figura B.2: Funcionamiento de una capa GRU.

Apéndice C

Capas de regularización

C.1. Dropout

El *Dropout* es el proceso de poner a cero aleatoriamente algunas unidades durante el proceso de entrenamiento, por ello, es una técnica de inyección de ruido. Esto crea numerosas redes más pequeñas que necesitan aprender a resolver la tarea original.

Como se observa en la figura C.1, durante cada pasada hacia adelante en el proceso de entrenamiento, los nodos se ponen a cero con una probabilidad, "p", que es un hiperparámetro. Esto implica que, en lugar de la salida normal de la función de activación, el nodo sólo produce ceros para esa pasada hacia delante. Dado que la reducción a cero de los nodos reduce el peso total de los valores de activación, los nodos restantes se incrementan en 1/p.

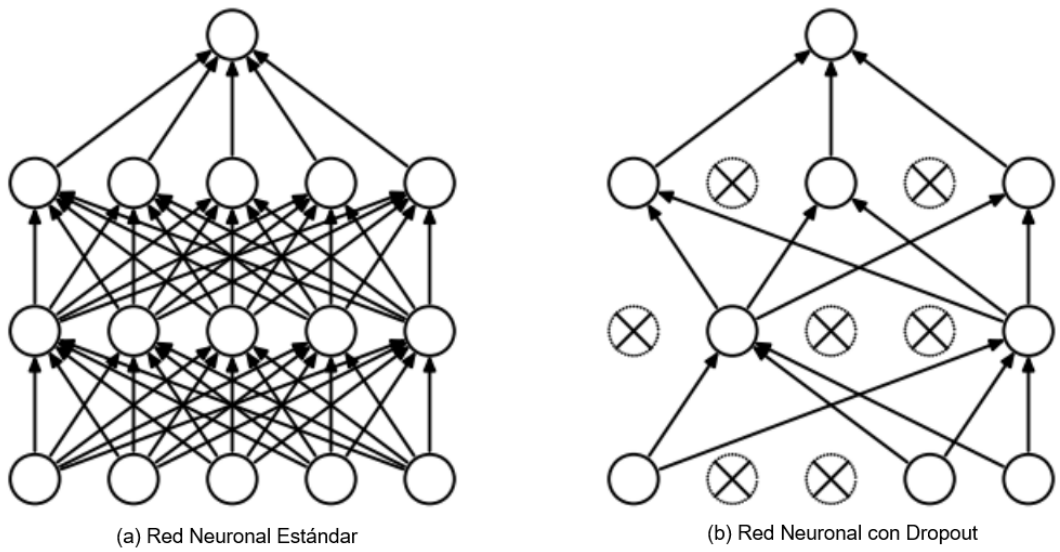

Figura C.1: Conexiones existentes en **a** una red neuronal estándar y **b** una red neuronal aplicando *Dropout*.

Durante las pruebas y la inferencia, se utiliza la red completa, sin ningún abandono. El efecto de los distintos abandonos durante el entrenamiento es como si se crearan muchas redes más pequeñas. Cuando se utiliza toda la red para las pruebas y la inferencia, equivale a tener un conjunto de estas redes más pequeñas, lo que reduce el sobreajuste.

C.2. Normalización de lotes

La capa de normalización por lotes funciona normalizando las activaciones de cada capa en mini lotes durante el entrenamiento. Esto se logra calculando la media y la desviación estándar de cada dimensión de las activaciones dentro de un lote y luego aplicando una transformación lineal para centrar y escalar estas activaciones. Esta normalización ayuda a reducir la covariación no deseada entre las activaciones, lo que facilita el entrenamiento de redes más profundas y permite que cada capa aprenda de manera más independiente. Además, la normalización por lotes puede actuar como una forma de regularización al introducir un poco de ruido en el proceso de entrenamiento. En la inferencia, las activaciones se normalizan utilizando las estadísticas calculadas durante el entrenamiento en lugar de las estadísticas de mini lotes.

C.3. Normalización de capa

La capa de normalización de capa funciona calculando la media y la desviación estándar de las activaciones de cada ejemplo a lo largo de todas las unidades en una capa y luego aplicando una transformación lineal para centrar y escalar estas activaciones. Esta técnica ayuda a estabilizar el proceso de entrenamiento al reducir la covariación entre las activaciones de las unidades dentro de una capa y mejora la capacidad de generalización del modelo. Durante la inferencia, las activaciones se normalizan utilizando las estadísticas calculadas durante el entrenamiento para cada ejemplo individualmente. De esta forma, la capa de normalización de capa mejora la estabilidad y el rendimiento de las redes neuronales al facilitar el entrenamiento y mejorar la capacidad de generalización del modelo.