Trabajo de Fin de Grado

Flogo: Un DSL para el diseño y gestión del ciclo de vida de una red neuronal

TITULACIÓN: Grado en Ciencia e Ingeniería de Datos

AUTOR: José Juan Hernández Gálvez

TUTORIZADO POR
José Évora Gómez
Octavio Roncal Andrés

Junio 2024

Agradecimientos

A mi madre. A mi padre.

Resumen

Este documento describe un lenguaje específico de dominio (DSL) diseñado para simplificar y automatizar el proceso de creación de redes neuronales. Su objetivo principal es abstraer la complejidad técnica de frameworks como PyTorch o TensorFlow. De esta manera, permite que los desarrolladores puedan modelar tanto la estructura de redes neuronales como su proceso de entrenamiento utilizando un lenguaje más intuitivo y directo. Además, incorpora un generador de código que transforma el modelo descrito con el DSL a código Python. Esta automatización acelera el proceso de desarrollo y mantenimiento, haciéndolo más eficiente. Además, simplifica el proceso de desarrollo para aquellos que no son expertos en redes neuronales, haciendo más comprensible así los modelos de redes neuronales y fomentando una mayor innovación. Esta accesibilidad y facilidad de uso permiten que más personas contribuyan y experimenten con proyectos de inteligencia artificial, impulsando el avance tecnológico en este área.

Abstract

This document describes a domain-specific language (DSL) designed to simplify and automate the process of creating neural networks. Its primary goal is to abstract the technical complexity of frameworks like PyTorch or TensorFlow. In this way, it allows developers to model both the structure of neural networks and their training process using a more intuitive and straightforward language. Additionally, it incorporates a code generator that transforms the model described with the DSL into Python code. This automation accelerates the development and maintenance process, making it more efficient. Furthermore, it simplifies the development process for those who are not experts in neural networks, thereby making neural network models more comprehensible and fostering greater innovation. This accessibility and ease of use enable more people to contribute to and experiment with artificial intelligence projects, driving technological advancement in this area.

Índice general

Índice de figuras

Capítulo 1

Introducción

> No se puede entender el buen diseño si no se entiende a las personas.
>
> Dieter Rams

Durante la formación académica que hemos recibido en la Universidad de Las Palmas de Gran Canaria, el uso de herramientas como *PyTorch* [30] y *TensorFlow* [16] ha sido habitual para la implementación de técnicas relacionadas con el aprendizaje de redes neuronales. Estas herramientas nos permiten la implementación de redes neuronales, proporcionando un entorno versátil operado a través de *Python*. No obstante, a pesar de su popularidad, *PyTorch* y *TensorFlow* introducen una serie de problemas. La definición de arquitecturas de modelos, la manipulación de tensores y la configuración de procesos de entrenamiento y validación requieren un nivel conocimiento considerable no solo de los principios de aprendizaje profundo, sino también de las peculiaridades de cada *Framework*. Este nivel de complejidad técnica a menudo se traduce en una barrera significativa para aquellos que no poseen una sólida base en programación y redes neuronales.

Además, la experiencia con estas herramientas nos ha permitido conocer sus limitaciones, particularmente en la necesidad de repetir código para tareas comunes y de la dificultad para comprender el código complejo asociado a las redes neuronales. La frustración al enfrentar largas sesiones de depuración y ajustes finos, nos llevó de forma natural a plantear la posibilidad de desarrollar un lenguaje que pudiera simplificar y automatizar operaciones recurrentes. Por un lado, configurar correctamente las redes neuronales, gestionar las dimensiones de los datos de entrada/salida, y ajustar los hiperparámetros puede ser tedioso y propenso a errores. Por otro lado, muchas tareas, como la preparación de datos, la regularización de modelos, y las pruebas de validación, se repiten con pequeñas variaciones entre proyectos, lo que sugiere una oportunidad para la automatización. Además, y aunque son potentes, *PyTorch* y *TensorFlow* requieren un entendimiento sólido de conceptos de programación y matemáticas, lo cual puede ser otra barrera para los principiantes. Esto concluye con que los estudiantes a menudo enfrentan una curva de aprendizaje significativa debido a la complejidad inherente de los conceptos y la estructura con la que se tiene que programar.

De esta experiencia compartida y de discusiones muy interesantes con otros dos compañeros de clase, surgió la idea de realizar un proyecto conjunto que pudiéramos realizar como parte de nuestros Trabajos de Fin de Título (TFT). El resultado de este proyecto colaborativo es un sistema llamado *Flogo* [17], que incluye el resultado de este proyecto: un lenguaje orientado específicamente al desarrollo de redes neuronales y un generador de código.

Este lenguaje, al centrarse en las necesidades específicas de los modelos de redes neuronales, ofrece notaciones más intuitivas y directas que las proporcionadas por un lenguaje como *Python*. También permite representar las abstracciones de forma más simple, reduciendo el código necesario y minimizando el potencial de errores. Además, este lenguaje tiene el potencial de acelerar el proceso de desarrollo y mantenimiento.

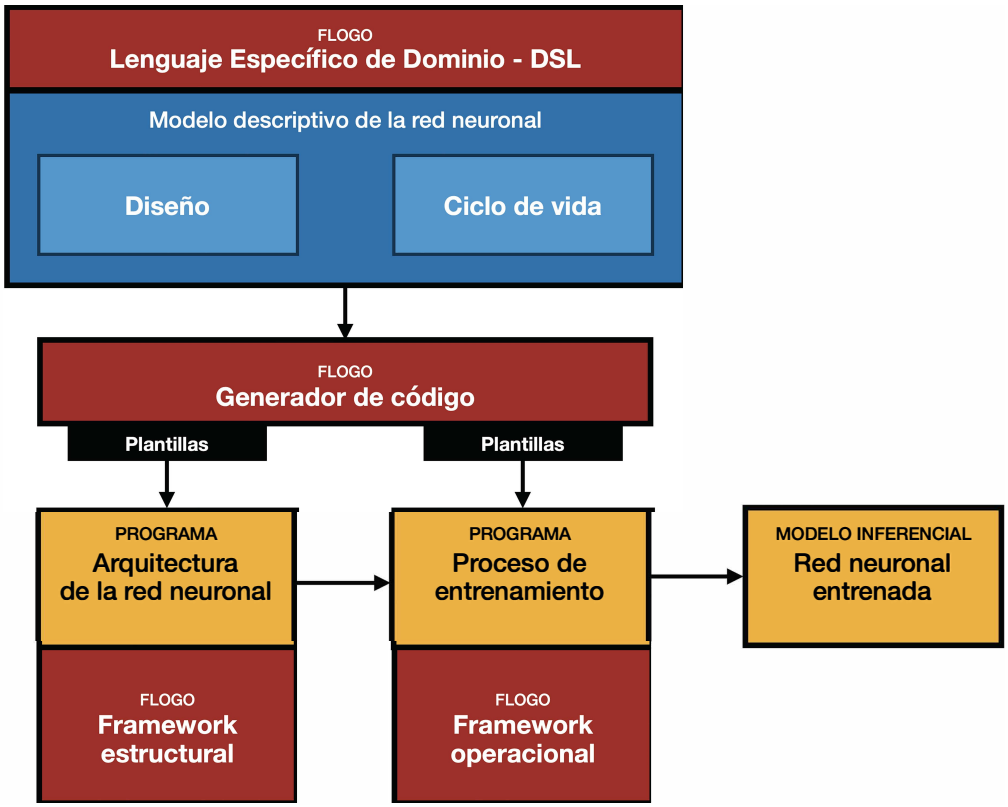

Ilustración 1.1: Diagrama del Sistema *Flogo*

La arquitectura general de *Flogo* que se puede ver en la figura 1.1 se ha planteado para facilitar la colaboración entre los miembros del equipo, permitiendo que cada uno se concentre en componentes distintos del sistema sin perder la cohesión y la integración del proyecto completo. Esto se ha logrado mediante una arquitectura que divide claramente las responsabilidades y facilita la integración de trabajo. Los componentes de *Flogo* son los siguientes:

- **Lenguaje de Flogo**. Es el componente principal que permite a los usuarios definir redes neuronales. Este se enfoca en dos aspectos
  - **Diseño**. Se refiere a la configuración de la arquitectura de la red neuronal, como

capas, funciones de activación, etc.

    - **Ciclo de vida**. Involucra la gestión del ciclo de vida de la red neuronal, incluyendo su creación, entrenamiento y evaluación.
- **Generador de código**. Toma las especificaciones realizadas con el lenguaje y genera el código necesario para implementar y entrenar la red neuronal.
- **Plantillas**. Son los componentes que se encargan de transformar el modelo descriptivo en código ejecutable, son utilizadas por el generador de código.
- **Frameworks**. Se han creado dos, cada uno con un propósito:
    - **Estructural**. Facilita el diseño y la configuración de la arquitectura de la red neuronal, incluyendo la definición de las capas, sus tipos, y las interconexiones entre ellas.
    - **Operacional**. Se enfoca en el proceso de entrenamiento y ajuste de la red neuronal. Este framework maneja las operaciones necesarias para entrenar el modelo y optimizar sus parámetros.

Este enfoque modular no solo optimiza la colaboración entre los integrantes, sino que permite que su arquitectura pueda adaptarse y evolucionar en respuesta a las necesidades tecnológicas y a los avances en el campo del aprendizaje profundo. Desde el punto de vista evolutivo, si bien *Flogo* se ha desarrollado inicialmente para generar código para *Python* y *PyTorch*, la visión es incluir soporte para otros lenguajes, como C++ o Java; o nuevos frameworks como TensorFlow, Microsoft Cognitive Toolkit [28] (CNTK) o Apache MXNet [1].

1.1. Lenguajes específicos de dominio

Los lenguajes de programación se pueden clasificar en lenguajes de propósito general (*GPL*) y lenguajes específicos de dominio (*DSLs*) [13]. Los *DSLs* están diseñados para ofrecer soluciones eficientes a problemas específicos, a diferencia de los lenguajes de propósito general que son más flexibles y abarcan una amplia gama de aplicaciones. Históricamente, los *DSLs* han sido denominados de diversas formas como lenguajes orientados a aplicaciones, de propósito especial, especializados, específicos de tarea o simplemente lenguajes de cuarta generación (*4GL*).

Los *DSLs* sacrifican la generalidad para mejorar la expresividad en dominios limitados. Proporcionan notaciones y constructos específicamente adaptados a un dominio particular, lo que se traduce en un aumento significativo de la expresividad y la facilidad de uso en comparación con los *GPLs*. Esto conlleva a ganancias sustanciales en productividad y una reducción en los costos de mantenimiento. Además, al requerir menos conocimiento experto en el dominio y en programación, los *DSLs* hacen accesible su dominio de aplicación a un grupo más amplio de desarrolladores de software.

Los *DSLs* no son un concepto nuevo en el ámbito de la computación. Desde los primeros días de la informática, se han desarrollado lenguajes diseñados para abordar problemas

específicos de dominio. Uno de los primeros ejemplos de un *DSL* es el lenguaje APT [5] (*Automatically Programmed Tool*), desarrollado en 1957-1958 para la programación de máquinas herramientas controladas numéricamente. APT simplificó la tarea de codificar las instrucciones para máquinas herramientas, permitiendo a los operadores concentrarse en los aspectos de diseño sin tener que manejar los complejos códigos de bajo nivel.

A lo largo de los años, los *DSLs* han encontrado aplicaciones en una amplia gama de campos, desde SQL, para la manipulación y consulta de bases de datos; LaTeX, un lenguaje de marcado utilizado para la preparación de documentos técnicos y científicos; o HLSL [26] (*High-Level Shading Language*) usado en desarrollo de videojuegos para gestionar efectos como iluminación, sombreado y color. Con el avance de la tecnología y la creciente complejidad de los sistemas software, los lenguajes específicos de dominio como el *HyperText Markup Language* (HTML) y el *Cascading Style Sheets* (CSS) han jugado un papel crucial en transformar el campo del desarrollo web. Estos lenguajes de marcado y estilo no solo han simplificado la creación y diseño de sitios web, sino que también han hecho que estas tareas sean accesibles a una audiencia mucho más amplia, incluyendo a personas sin una formación técnica avanzada en programación.

En muchos casos, el potencial de un *DSL* para mejorar el desarrollo dentro de un dominio específico no es inmediatamente reconocible. Los desarrolladores pueden comenzar utilizando *GPLs* para abordar problemas dentro de un dominio particular debido a la familiaridad y la accesibilidad de estos lenguajes. Sin embargo, a medida que el desarrollo avanza y los desafíos específicos del dominio se vuelven más claros, la necesidad de un enfoque más especializado y eficiente puede empezar a ser evidente. Es decir, a medida que los desarrolladores se enfrentan a la creciente complejidad o a las necesidades especializadas de su campo, pueden comenzar a conceptualizar cómo un *DSL* podría resolver estos problemas de manera más efectiva. Este reconocimiento puede surgir de la frustración con las limitaciones de los *GPLs* o de la identificación de patrones repetitivos y tareas que podrían ser automatizadas o simplificadas mediante un *DSL*.

Este proceso evolutivo sugiere que el diseño de *DSLs* debe ser iterativo y receptivo a las necesidades emergentes del dominio. También se destaca la importancia de la colaboración entre expertos en dominios específicos y expertos en lenguajes de programación para crear *DSLs* que sean verdaderamente efectivos y eficientes.

1.2. Ingeniería de software de redes neuronales

A pesar de los impresionantes avances y aplicaciones del Deep Learning, la ingeniería de las redes neuronales a menudo se enfrenta a desafíos que impiden su mantenimiento eficiente. Considerando que las redes neuronales son un tipo de software, el mantenimiento de estos sistemas, ya sea correctivo, adaptativo o evolutivo, es esencial para asegurar su funcionamiento óptimo a lo largo del tiempo.

Como se puede ver en la ilustración 1.2, inicialmente, el coste aumenta durante el desarrollo. En la fase de producción, el coste disminuye pero puede aumentar debido a la deuda técnica. Durante el mantenimiento, el coste se mantiene bajo hasta que el diseño se vuelve

obsoleto, lo que puede incrementar los costes nuevamente para actualizaciones o rediseños.

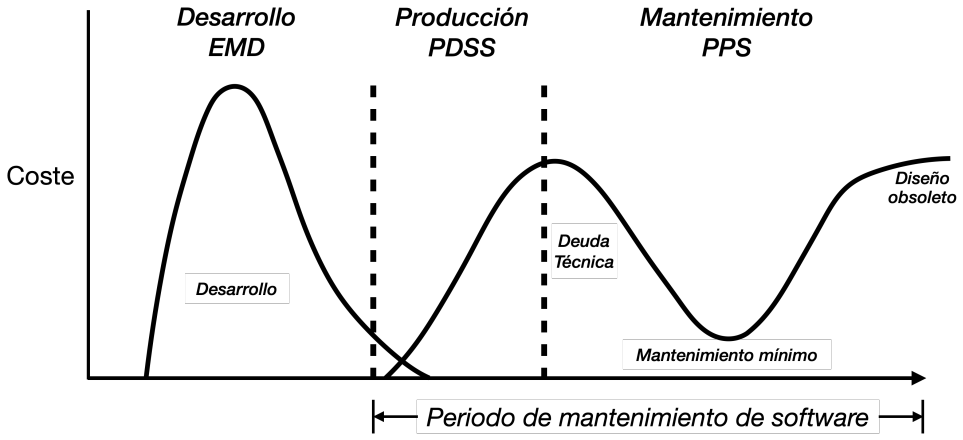

Ilustración 1.2: Costes del Software.

El mantenimiento correctivo se refiere a las modificaciones realizadas en el software para corregir problemas identificados después de la implementación. En el contexto de las redes neuronales, esto puede incluir: corrección de errores, mejora de la precisión del modelo. Este tipo de mantenimiento es reactivo; es decir, ocurre en respuesta a problemas identificados y suele ser imprevisto, lo que puede resultar en costos y esfuerzos significativos dependiendo de la complejidad del error y la profundidad de la integración del sistema.

El mantenimiento adaptativo implica modificar el sistema para que siga siendo operativo y relevante frente a los cambios en su entorno operativo. Esto puede incluir: actualización de plataformas y dependencias o a adaptación a nuevos requisitos regulatorios. Este tipo de mantenimiento es necesario para asegurar que el sistema pueda continuar funcionando en un entorno cambiante e implica revisiones regulares para anticipar y responder a estas necesidades.

El mantenimiento evolutivo se enfoca en extender o mejorar el sistema para satisfacer necesidades emergentes y aprovechar nuevas oportunidades tecnológicas. Incluye agregar nuevas funcionalidades y optimización para mejorar el rendimiento. Este tipo de mantenimiento es proactivo y está dirigido por los cambios en los requisitos del usuario y por las oportunidades de mejora identificadas por los desarrolladores o por los comentarios de los usuarios.

Es cierto que los procesos de mantenimiento en sistemas que incorporan redes neuronales presentan desafíos únicos, debido en parte a la complejidad inherente de estos sistemas. Sin embargo, es fundamental reconocer que esta complejidad no se debe exclusivamente a las redes neuronales, sino que también involucra múltiples factores como la integración de sistemas, la gestión de datos y la interacción con otras tecnologías. Aunque las redes neuronales añaden una capa de sofisticación técnica, el objetivo es diseñar y emplear estos sistemas de manera que minimicen la complejidad adicional que podrían introducir [14].

Esto se logra a través de la mejora del modelado y la implementación de prácticas de desarrollo que aseguren mantenibilidad y escalabilidad, sin sobrecargar el sistema con complejidades. Por un lado, la complejidad esencial que es inherente al problema mismo que el

software intenta resolver. No puede ser eliminada, simplificada o reducida por mejoras en el proceso de desarrollo de software, herramientas o tecnología, porque está directamente ligada a la naturaleza del problema. Por otro lado, la complejidad accidental se refiere a la complejidad que no es necesaria para resolver un problema específico, sino que surge de la manera en que las soluciones son implementadas. A diferencia de la complejidad esencial, la complejidad accidental puede ser mitigada o eliminada mediante la selección de mejores herramientas, técnicas o procesos de desarrollo.

El desafío para los desarrolladores de redes neuronales es maximizar la reducción de la complejidad accidental sin agravar la complejidad esencial. Sin embargo, las prácticas comunes en la implementación de redes neuronales a menudo introducen complejidad accidental. En muchos proyectos de redes neuronales, el código puede volverse extremadamente complejo y entrelazado, especialmente bajo presión para mejorar el rendimiento y la precisidad del modelo. Esto conduce a una "programación espagueti" donde el código es difícil de leer, mantener y actualizar. No obstante, hay otros problemas adicionales como:

- **Dependencia de plataformas específicas**. Muchas implementaciones de redes neuronales están fuertemente acopladas a *frameworks* y bibliotecas específicas, lo que limita su portabilidad y adaptabilidad a nuevas plataformas o tecnologías emergentes, y por tanto incumpliendo el *Principio de sustitución de Liskov*[25].
- **Falta de modularidad**. Las redes neuronales complejas a menudo se construyen como bloques monolíticos con pocas interfaces claras entre los componentes del sistema. Esto dificulta la recombinación de código y complica la prueba de componentes individuales.
- **Riesgo de obsolescencia técnica**. La rapidez con la que evoluciona el campo del *Deep Learning* puede hacer que los modelos y herramientas se vuelvan obsoletos rápidamente, lo que obliga a los equipos de desarrollo a actualizar constantemente sus sistemas para mantener su relevancia y efectividad.
- **Falta de documentación**. La rápida evolución de las técnicas de deep learning y la experimentación constante en este campo a menudo resultan en una falta de documentación adecuada. Esto puede dejar a los proyectos sin el soporte necesario para su mantenimiento y escalabilidad a largo plazo.

Tradicionalmente, la ingeniería de software se ha apoyado en principios bien establecidos, diseñados para controlar la complejidad. El problema es que la formación de muchos desarrolladores de redes neuronales, aunque robusta en matemáticas, estadísticas y conocimientos específicos de aprendizaje automático, a menudo adolece de una sólida base en principios de ingeniería de software. Este déficit en la formación puede tener implicaciones en el desarrollo, la implementación y el mantenimiento de sistemas basados en inteligencia artificial.

Una estrategia prometedora para abordar estas deficiencias implica el uso y desarrollo de *DSLs*. Los *DSLs* proporcionan abstracciones de alto nivel que simplifican aspectos complejos del desarrollo de redes neuronales[13]. Al ofrecer construcciones específicas del dominio, estos lenguajes pueden ocultar la complejidad subyacente y reducir la carga cognitiva sobre los desarrolladores, permitiéndoles concentrarse en la lógica del problema más que en los detalles de implementación. Además, al proporcionar un lenguaje común basado en los principios del dominio, los *DSLs* facilitan una mejor colaboración entre los miembros del equipo, incluidos

aquellos que pueden no tener una formación profunda en programación. Esto es especialmente útil en equipos interdisciplinarios donde los expertos en el dominio necesitan trabajar mano a mano con ingenieros de software.

Por otro lado, al estandarizar las tareas de desarrollo comunes mediante un *DSL*, se puede asegurar que las prácticas óptimas sean aplicadas de manera uniforme. Esto ayuda a prevenir errores comunes y mejora la coherencia y calidad del código producido, facilitando su mantenimiento y escalabilidad.

Los *DSLs* pueden ser diseñados para ser intuitivos para los especialistas del dominio, lo que facilita la documentación y el mantenimiento del software. Al utilizar un lenguaje que refleje directamente los conceptos del dominio, se vuelve más fácil para los nuevos desarrolladores entender y trabajar con el código existente. Con ello, los *DSLs* pueden aumentar significativamente la productividad al reducir la cantidad de código que los desarrolladores necesitan escribir y al minimizar el potencial de errores. Esto es particularmente valioso en un campo que evoluciona rápidamente como el de las redes neuronales, donde la capacidad de iterar rápidamente y probar nuevas ideas es crucial.

1.3. Estructura de la memoria

La memoria se estructura de la siguiente manera:

- **Diseño del lenguaje**. Profundiza en los constructos del lenguaje específico de dominio (*DSL*) que se han identificado.
- **Implementación**. Se discute el proceso que nos ha permitido implementar el lenguaje y la generación de código.
- **Ejemplos**. Se muestran aplicaciones prácticas de *Flogo* a través de estudios de caso o ejemplos concretos.
- **Proceso de desarrollo**. Detalla la metodología y herramientas utilizada en el proyecto.
- **Conclusiones y trabajo futuro**. Resume los resultados principales y las principales aportaciones. Además, se señalan direcciones para la investigación, sugiriendo áreas en las que el trabajo actual podría expandirse o mejorarse.

Es importante señalar que en el campo de la inteligencia artificial es común usar términos en inglés debido a su amplia adopción y falta de traducción. Por ello, se informa a los lectores que se utilizarán varios anglicismos, que estarán explicados en detalle en el glosario.

Capítulo 2

Estado actual y objetivos iniciales

> No es suficiente para el código, que funcione.
>
> Robert C. Martin

2.1. Estado actual

Las redes neuronales son un pilar fundamental en el campo de la inteligencia artificial (IA), modelando relaciones y patrones complejos en datos de manera similar al cerebro humano [27]. Estas redes están compuestas por capas de nodos interconectados, donde cada nodo representa una neurona artificial. A través del entrenamiento, las redes neuronales aprenden ajustando pesos en respuesta a los datos de entrada, mejorando así su precisión en tareas como clasificación, regresión, y generación de datos.

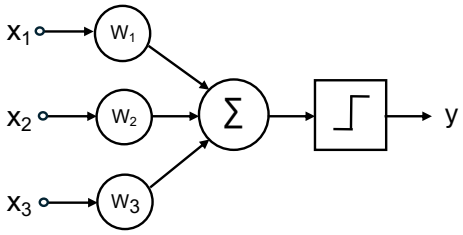

Ilustración 2.1: Esquema de un Perceptrón.

El *Deep Learning* (DL) como disciplina comenzó en los años 40 con la neurona de McCulloch y Pitts [27]. Posteriormente, en los años 50, surgió el perceptrón de Frank Rosenblatt [12], el cual se puede observar en la ilustración 2.1. El perceptrón, estableció la base para los futuros desarrollos en redes neuronales, es la unidad básica de la red neuronal actual. Consiste en entradas $(x_1, x_2, \ldots, x_n)$, pesos $(w_1, w_2, \ldots, w_n)$ y una función de activación. Las entradas se multiplican por los pesos y se suman, pasando luego a través de una función de activa-

ción para producir la salida final. Sin embargo, debido a sus limitaciones en la capacidad de comprensión de modelos más complejos, el interés en las redes neuronales decayó durante las décadas siguientes, comenzando lo que se denomina el invierno de la IA (*AI Winter*).

En los años 80, el interés se reavivó gracias a la introducción del algoritmo de *Back Propagation* por Rumelhart, Hinton y Williams [35], que permitió entrenar redes con múltiples capas ocultas. Esto fue crucial para el aprendizaje de representaciones internas más complejas. Sin embargo, los problemas de sobreajuste y la falta de capacidad computacional limitaron su aplicación práctica.

La era moderna del *Deep Learning* comenzó a principios del siglo XXI, impulsada por dos factores principales: el aumento masivo en la capacidad de cómputo, especialmente debido a la introducción de las GPUs, y la disponibilidad de grandes conjuntos de datos, como ImageNet [8]. Estos factores permitieron entrenar redes neuronales profundas (*Deep Neural Networks*) que eran mucho más grandes y complejas que antes.

En 2012, Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton demostraron los poderosos efectos de las redes neuronales profundas en el reconocimiento de imágenes con su modelo AlexNet [21], que ganó la competición ImageNet por un margen significativo. Este evento marcó el comienzo de la "era del deep learning", donde las redes profundas comenzaron a ser aplicadas con éxito no solo en visión por computadora sino también en otras áreas como el procesamiento de lenguaje natural y el reconocimiento de voz.

El *Deep Learning* ha continuado evolucionando y transformando el campo de la inteligencia artificial. Modelos como GPT-3 [6] para generación de texto, BERT [10] para procesamiento de lenguaje natural, y AlphaFold [20] para la predicción de estructuras de proteínas, son testimonios de cómo el *Deep Learning* puede abordar problemas de una complejidad que antes parecía inalcanzable.

A continuación, se procederá a explicar distintos tipos de redes neuronales existentes y las herramientas más utilizadas en la actualidad para su desarrollo y gestión. Por último, se discutirá la ingeniería de redes neuronales que se hace hoy en día, que están definiendo el avance y la aplicación de estas tecnologías.

2.1.1. Tipos de redes neuronales

Existen muchos tipos de redes neuronales, cada uno diseñado con estructuras y algoritmos específicos para abordar diferentes tipos de problemas en el campo de la inteligencia artificial. Estas variaciones en la arquitectura permiten que las redes neuronales sean increíblemente versátiles, adaptándose desde el análisis de datos visuales hasta el procesamiento de lenguaje natural y más allá. A continuación, exploraremos algunos de los tipos más destacados de redes neuronales [15] que se utilizan en la investigación y aplicaciones prácticas.

- **Redes Neuronales Feedforward (FNN)**[2]: Las redes neuronales Feedforward son el tipo más básico de redes neuronales. En estas redes, la información fluye en una sola dirección desde la entrada hacia la salida, pasando por una o más capas ocultas sin ciclos ni conexiones de retroalimentación. Este diseño simple las hace ideales para

tareas de clasificación y regresión básicas.

- **Redes Neuronales Convolucionales (CNN)**[22]: Especialmente diseñadas para procesar datos estructurados en forma de matriz, como las imágenes, estas redes han revolucionado el campo del análisis y procesamiento de imágenes. Utilizan un proceso conocido como convolución que filtra la entrada para extraer características importantes, lo que las hace extremadamente efectivas para tareas como el reconocimiento de imágenes y vídeo.
- **Redes Neuronales Recurrentes (RNN)**[19]: Las redes neuronales recurrentes son clave para manejar datos secuenciales como el texto o las series temporales. A diferencia de las FFNN, las RNN tienen conexiones de retroalimentación que les permiten mantener un estado o memoria sobre las entradas anteriores. Esto es vital para tareas donde el contexto o la secuencialidad de los datos importa, como en la traducción automática o en la predicción de series temporales.
- **Transformers**[37]: Los Transformers son una arquitectura de redes neuronales que ha revolucionado el campo del procesamiento del lenguaje natural (NLP). Este modelo se basa en el mecanismo de atención, que permite ponderar la importancia relativa de diferentes partes de la entrada, como palabras en una oración.

2.1.2. Herramientas actuales

En el desarrollo de redes neuronales, las herramientas y bibliotecas disponibles juegan un papel crucial al proporcionar las capacidades necesarias para diseñar, entrenar y desplegar modelos. Dos de las bibliotecas más prominentes en la comunidad de son *PyTorch* y *TensorFlow*. Ambas ofrecen amplias funcionalidades que facilitan desde investigaciones académicas hasta aplicaciones industriales a gran escala.

Ambos *frameworks* están respaldados por grandes comunidades de desarrolladores y constante desarrollo y mantenimiento, garantizando una gran cantidad de recursos de aprendizaje, soporte y actualizaciones continuas. Esto los distingue de otras opciones como *Microsoft Cognitive Toolkit* o *Apache MXNet*, que aunque potentes, no ofrecen el mismo nivel en términos de investigación y adaptabilidad de producción. Además, la amplia adopción de *TensorFlow* y *PyTorch* facilita la colaboración y el intercambio de conocimientos, haciéndolos herramientas estratégicas para cualquier proyecto que busque mantenerse al frente en la innovación de redes neuronales.

Además estos dos *frameworks* son de código abierto, lo cual fomenta una gran colaboración entre investigadores y desarrolladores de todo el mundo, acelerando la innovación y el progreso en el campo. Esto también permite que las empresas y los académicos las personalicen a su gusto sin depender de soluciones propietarias.

2.2. Objetivos iniciales

Este proyecto tiene como objetivo principal simplificar y automatizar el ciclo de vida completo de las redes neuronales, desde su concepción hasta su implementación final mediante la creación de un *DSL*.

La visión consiste en facilitar el acceso a la tecnología avanzada de aprendizaje profundo, proporcionando herramientas que abstraigan la complejidad de uso de *frameworks* como *Pytorch* o *TensorFlow*. Usando su *DSL*, *Flogo* permite a los usuarios modelar redes neuronales con notaciones intuitivas que no requieren un profundo conocimiento técnico en programación. Esto facilita una participación más amplia y diversa en el campo del aprendizaje automático, promoviendo la innovación y la colaboración en diversos sectores industriales y académicos. No obstante, este enfoque no solo pretende hacer más accesible la tecnología de redes neuronales para no especialistas, sino también simplificar el proceso de desarrollo para investigadores y desarrolladores experimentados.

Con la implementación de *Flogo*, se plantean como objetivos específicos dar soporte a las necesidades y desafíos actuales en el desarrollo de sistemas de aprendizaje automático:

- **Automatización y simplificación**: Crear un *DSL* que permita a los usuarios definir de manera sencilla y clara la arquitectura de una red neuronal y su proceso de entrenamiento y validación. Este lenguaje deberá abstraer los detalles técnicos y proporcionar una interfaz amigable para usuarios de diferentes niveles de habilidad técnica.
- **Generación de código eficaz**: Desarrollar un sistema de generación de código que convierta las especificaciones del *DSL* en código ejecutable. Este sistema deberá ser capaz de construir automáticamente la estructura de la red y configurar el pipeline de entrenamiento y validación, optimizando así el tiempo de desarrollo y reduciendo la posibilidad de errores manuales.
- **Optimización de procesos**: Implementar mecanismos para el entrenamiento eficiente y la validación rigurosa de los modelos generados, asegurando que estos sean robustos y confiables antes de su despliegue en entornos reales.

2.3. Competencias específicas

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.

EC1: Capacidad para comprender la importancia de la negociación, los hábitos de trabajo efectivos, el liderazgo y las habilidades de comunicación en todos los entornos de desarrollo de software.

EC3: Conocimiento y aplicación de los principios fundamentales y técnicas básicas de los

sistemas inteligentes y su aplicación práctica.

EC4: Conocimiento y aplicación de los principios, metodologías y ciclos de vida de la ingeniería del software.

EC5: Capacidad para diseñar y evaluar interfaces persona computador que garanticen la accesibilidad y usabilidad a los sistemas, servicios y aplicaciones informáticas.

ET1: Capacidad para comprender el entorno de una organización y sus necesidades en el ámbito de las tecnologías de la información y las comunicaciones.

ET2: Capacidad para seleccionar, desplegar, integrar y gestionar sistemas de información que satisfagan las necesidades de la organización, con los criterios de coste y calidad identificados.

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.

ED2: Capacidad para adquirir, obtener, formalizar y representar el conocimiento humano en una forma computable para la resolución de problemas mediante un sistema informático en cualquier ámbito de aplicación, particularmente los relacionados con aspectos de computación, percepción y actuación en ambientes o entornos inteligentes

ED3: Capacidad para conocer y desarrollar técnicas de aprendizaje computacional y diseñar e implementar aplicaciones y sistemas que las utilicen, incluyendo las dedicadas a extracción automática de información y conocimiento a partir de grandes volúmenes de datos.

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.

ED5: Capacidad para seleccionar, diseñar, desplegar, integrar, evaluar, construir, gestionar, explotar y mantener las tecnologías de hardware, software y redes para dar soporte a aplicaciones en Ciencia e Ingeniería de Datos.

ED6: Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos utilizados en Ciencia de Datos, particularmente las relacionadas con el análisis, predicción y prospectiva de grandes volúmenes de datos.

ED7: Capacidad para participar activamente en la especificación, diseño, implementación y mantenimiento de los sistemas de información y de las comunicaciones de uso común en Ciencia e Ingeniería de Datos y de diseñar e implementar software de aplicaciones para estos sistemas.

ED8: Capacidad de concebir sistemas, aplicaciones y servicios basados en tecnologías de la información y las comunicaciones para Ciencia e Ingeniería de Datos, incluyendo Internet, web, comercio electrónico, multimedia, servicios interactivos y computación móvil.

ED9: Capacidad para definir en la empresa problemas del dominio de Ciencia e Ingeniería de Datos y trasladar los análisis estadísticos a actuaciones de Inteligencia de Negocios conducidas por los datos para mejorar el rendimiento.

Capítulo 3

Diseño del lenguaje

> Cualquiera puede hacer código que el ordenador pueda entender. Los buenos programadores escriben código que los humanos pueden entender.
>
> Martin Fowler

Un lenguaje de programación está orientado por *constructos*, que son las elementos conceptuales que permiten a los programadores la especificación de acciones y la definición de estructuras de datos. Los constructos en un lenguaje de programación suelen incluir variables, tipos de datos, operadores, estructuras de control (como bucles y condicionales), funciones y procedimientos, entre otros. Los constructos constituyen el marco analítico con el que el programador se enfrenta a la tarea de describir un software.

Este capítulo se estructura en varias secciones que abarcan el desarrollo de redes neuronales usando el *DSL* de *Flogo*. Se comienza con el marco analítico 3.1, donde se introduce el diseño estructural y el entrenamiento de redes mediante los constructos de "laboratorio" y "arquitectura", en los que más adelante, se detallarán sus respectivos componentes. Finalmente, la sección de gramática 3.2 expone el léxico 3.2.1, la sintaxis 3.2.2 y la semántica 3.2.3 de *Flogo*, permitiendo a los usuarios entender y adaptar el lenguaje a sus necesidades específicas.

3.1. Marco analítico

En el contexto específico de *Flogo*, utilizado para el desarrollo de modelos de redes neuronales, se introducen dos constructos esenciales. Por un lado, el constructo de "laboratorio" se centra en cómo se entrena la red neuronal. Cubre el número de épocas, la elección de la función de pérdida o de del optimizador. Por otro lado, la "arquitectura" se refiere al diseño estructural de la red neuronal, contiene la forma en que los datos fluyen a través de la red. A continuación veremos cada uno de ellos en detalle.

3.1.1. Laboratorio

Este constructo ofrece a los usuarios las herramientas para describir como se deben entrenar las arquitecturas de una forma estructurada y sistemática, promoviendo la innovación y la mejora continua en el desarrollo de soluciones de inteligencia artificial. Un laboratorio contiene los siguientes constructos:

- **Optimizer**: Este es el algoritmo que se encarga de ajustar los parámetros de la red neuronal mediante pequeños pasos o steps, fluyendo a través del espacio de búsqueda.
- **Loss Function**: Es el componente que se encarga de definir el espacio de búsqueda, por el que el modelo intentará llegar a un mínimo.
- **Dataset**: Aquí se define el conjunto de datos con el que se entrenará la red, incluyendo la capacidad de realizar un *split* para dividir los datos en conjuntos de entrenamiento, validación y prueba.
- **Strategy**: Se refiere al enfoque de entrenamiento que se tomará dependiendo del tipo de problema. Actualmente hay dos:
  - Regression Strategy: Cuando es una única salida.
  - Classification Strategy: Cuando son múltiples salidas.
- **Early Stopper**: Este componente se encarga de evitar el overfitting, deteniendo el entrenamiento cuando ya no se observan mejoras en la función de pérdida.
- **Check Point Saver**: Permite guardar el estado del modelo en puntos específicos donde se observa un buen rendimiento. Sin necesidad de estar reentrenando el modelo si en algún punto futuro del entrenamiento se observa una bajada en el rendimiento del modelo.

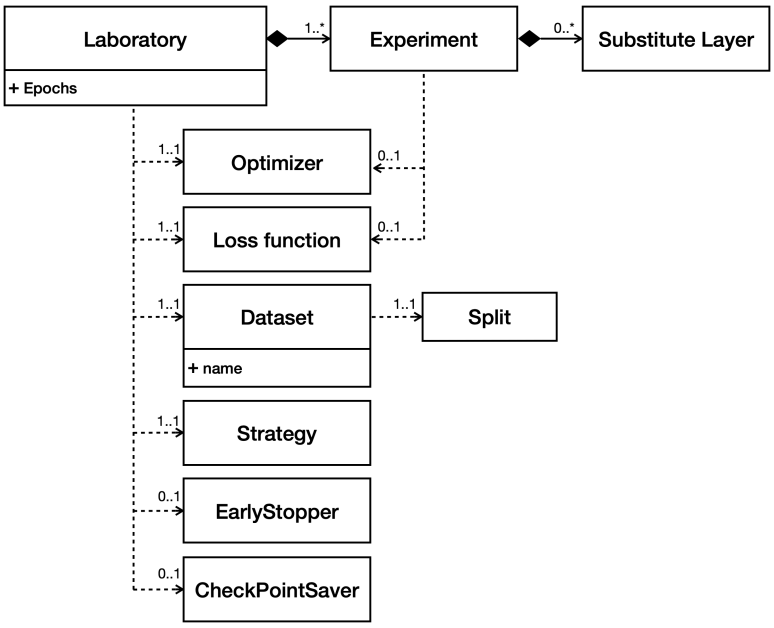

Ilustración 3.1: Constructos del laboratorio en UML

3.1.2. Arquitectura

Dentro de la arquitectura de la red neuronal, se han introducido tres conceptos estructurales fundamentales para facilitar el diseño y la comprensión de las redes neuronales complejas: las **secciones**, que están compuestas por **bloques**, y a su vez, los bloques están formados por **capas**. Estos elementos permiten a los usuarios construir arquitecturas de manera modular y limpia.

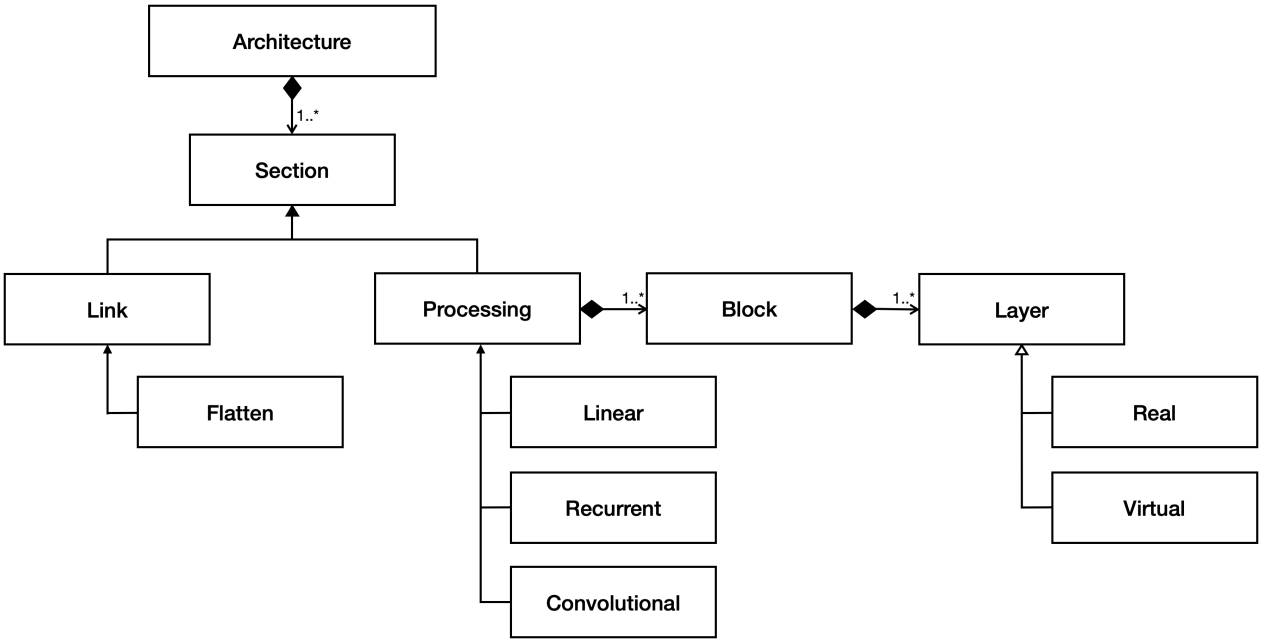

Ilustración 3.2: Constructos de la arquitectura en UML

3.1.3. Secciones

Dentro de la arquitectura de la red, las secciones se clasifican en dos categorías principales que facilitan la construcción y la funcionalidad de las redes neuronales: las **secciones de procesamiento** y las **secciones de enlace**. Cada una tiene un propósito específico en el diseño de la red neuronal.

Las secciones de procesamiento son aquellas diseñadas para contener capas o bloques que realizan transformaciones directas en los datos. Estas secciones son el núcleo donde se aplican las operaciones de mapeo necesarias para extraer características y aprender de los datos. Dentro de las secciones de procesamiento, actualmente encontramos tres subcategorías:

- **Secciones lineales**: Utilizadas para secuencias de capas, adecuadas para tareas de clasificación o regresión estándar. En estas secciones se trabaja con tensores de una dimensión.
- **Secciones recurrentes**: Adecuadas para tareas con dependencias temporales o secuenciales. En estas secciones se trabaja con tensores dos dimensiones y una dimensión.

- **Secciones convolucionales**: Orientadas a tareas de procesamiento visual, donde las operaciones convolucionales son primordiales. En estas secciones se trabaja con tensores de tres dimensiones.

Estas secciones están diseñadas para ser los pilares donde se definen y ejecutan los principales cálculos y algoritmos de la red.

Es importante destacar que la primera sección de cualquier arquitectura siempre es una Sección de Procesamiento, y es en esta donde se define el tamaño de entrada o *Input* de la red. Este concepto establece las dimensiones de entrada de la arquitectura.

Por otro lado, las secciones de Enlace son estructuras diseñadas específicamente para conectar secciones de Procesamiento. Su propósito principal es facilitar la comunicación y el traspaso de información entre secciones que pueden operar en diferentes dimensiones o niveles de abstracción. Esto es crucial para construir redes neuronales complejas donde diferentes secciones necesitan interactuar eficazmente sin pérdida de información relevante. Actualmente están las **secciones *flatten***, que tienen como objetivo transformar tensores de múltiples dimensiones tensores de una sola dimensión. Esto es especialmente útil cuando después de una sección convolucional o recurrente se quiere pasar a una sección lineal, ideal para hacer clasificaciones o regresiones.

3.1.4. Bloques

En *Flogo*, los bloques son las unidades fundamentales de las secciones de procesamiento al proporcionar una estructura modular que agrupa patrones de capas que comúnmente se repiten en las arquitecturas de redes neuronales. Los bloques no solo facilitan la construcción de la red, sino que también mejora significativamente la legibilidad y el mantenimiento del código.

Específicamente, dentro de las secciones lineales y convolucionales, los bloques pueden configurarse para que funcionen como **residuales**. Esto permite que los bloques no sólo procesen la información de manera secuencial, sino que también se creen conexiones residuales, cuando se requiera que un bloque pueda saber lo que ha realizado el bloque anterior. Los bloques residuales introducen un constructo nuevo: ***Shortcut***,

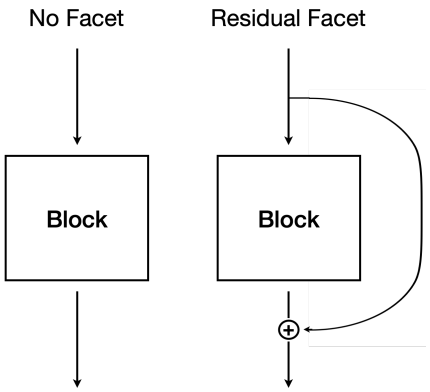

Ilustración 3.3: Facetas de los bloques en UML

El *Shortcut* es el procesamiento por el que la entrada del bloque pasa a la salida, para ser sumada con la salida. El *Shortcut* se puede configurar de dos maneras, dependiendo de las necesidades específicas del modelo y de la tarea que se esté ejecutando:

- **Default**, añade simplemente una capa simple, que puede ser convolucional o lineal según el contexto de la sección en la que se encuentra el bloque. Esta capa tiene como objetivo realizar una transformación básica de la entrada, proporcionando un camino más directo y menos complejo para el flujo de datos. Esto es especialmente útil para mantener la eficiencia del modelo y evitar la sobrecarga computacional sin comprometer la integridad del aprendizaje.
- **Custom**, permite una personalización detallada del conjunto de capas por las cuales pasará la entrada. Este enfoque ofrece la flexibilidad de definir transformaciones más complejas o específicas que pueden ser necesarias para adaptarse a características particulares de los datos o para abordar problemas específicos más eficazmente. Además, en la configuración *Custom*, no es necesario que la salida del *Shortcut* coincida con la salida del bloque principal en dimensiones, ya que se añade una capa adicional al final del *Shortcut* para ajustar las dimensiones según sea necesario para realizar la suma con la salida del bloque principal.

3.1.5. Capas

Las capas son los componentes de los bloques. Hay tres tipos de capas: reales, tanto de procesamiento como de activación, y virtuales.

- **Reales**: Son aquellas que realizan transformaciones efectivas y directas sobre los datos. Estas se subdividen en dos tipos:
  - **De procesamiento**: Estas capas son responsables de realizar las operaciones o mapeos específicos, son las encargadas del procesamiento de los tensores. Una característica fundamental en la definición de estas capas es el concepto de **Output**. Todos los tipos de capas de procesamiento están descritos en el anexo B.1
  - **De activación**: Se utilizan para introducir no linealidades en el modelo, lo que

permite a la red aprender y modelar relaciones complejas entre los datos. Su principal función es transformar la entrada lineal recibida de otras capas (como las capas de procesamiento) en una salida no lineal, lo que es crucial para tareas que requieren una capacidad de generalización más allá de simples transformaciones lineales.

- **Virtuales**: Las capas virtuales (*VLayers*) añaden flexibilidad y potencia a *Flogo*, particularmente durante la fase de experimentación. Aunque éstas no realizan operaciones en sí, son fundamentales para la parametrización de las arquitecturas. Permiten a los programadores especificar puntos dentro de la arquitectura que pueden ser dinámicamente reemplazados por capas reales en función de las materializaciones definidas en los experimentos del laboratorio.

Esta parametrización es esencial, ya que dota al proceso de diseño de modelos de la capacidad para adaptar y afinar las arquitecturas de forma programática, permitiendo la reutilización de estructuras con diferentes configuraciones de capas según se requiera.

Gracias a las capas virtuales se conciben las **arquitecturas colapsadas**. Este concepto se inspira en la Física Cuántica, donde la Función de Onda de un átomo colapsa en un estado observable al ser medida. De manera análoga, las capas virtuales junto con las materializaciones definidas en los experimentos 'colapsan' las arquitecturas parametrizables.

El constructo *Output* permite especificar directamente el tamaño de salida de la capa. Este enfoque simplifica significativamente el diseño de la red al permitir a los desarrolladores no tengan que estar enfocándose en ajustar estos parámetros, que puede resultar muy complejo.

El uso de este constructo es especialmente ventajoso en capas que alteran las dimensiones del tensor de entrada, como las capas convolucionales o las capas de agrupamiento (*Pooling*). Al definir directamente el tamaño de salida deseado, los diseñadores pueden asegurar que la red se adapte a los requisitos específicos del problema de manera más intuitiva y directa. Esto no solo facilita la configuración inicial de la red, sino que también hace que el proceso de ajuste y modificación de la red sea más eficiente, ya que los cambios en la arquitectura pueden implementarse rápidamente redefiniendo los valores de *Output* en las capas afectadas. El concepto de *Output* hace que el diseño de redes neuronales sea más flexible y fácil de configurar, permitiendo a los usuarios concentrarse en los aspectos más estratégicos del modelo, como la arquitectura general y el rendimiento, mientras se minimizan los aspectos técnicos y operativos más tediosos.

3.2. Gramática

Los componentes de cualquier lenguaje son léxico 3.2.1, sintáctico 3.2.2 y semántico 3.2.3. El léxico comprende una serie de palabras clave y símbolos que forman la base del vocabulario utilizado en *Flogo*. Estas palabras clave incluyen términos fijos y extensibles que permiten a los usuarios adaptar el lenguaje a tecnologías emergentes y necesidades específicas de modelado. La sintaxis define las estructuras que permiten componer las declaraciones del modelo. La sintaxis permite describir la arquitectura y la configuración de entrenamiento. Finalmente, la

semántica consiste en un conjunto de reglas que restringen las estructuras sintácticas, asegurando que las declaraciones del modelo sean coherentes. Estas reglas facilitan la traducción a los *frameworks* que dan soportan a la traducción.

3.2.1. Léxico

El léxico está formado por símbolos, y palabras clave fijas y extensibles. Los símbolos son NL, INDENT, DEDENT, '(', '), ',', '=' y 'as'. Se utilizan como conectores de las distintas estructuras del lenguaje. Las palabras clave fijas son 'Architecture', 'Block', 'Dataset', 'Experiment', 'Laboratory', 'Materialization' y 'Split'. Representan constructos que no requieren ser extendidos. Por último las palabras clave extensibles, indicadas en la gramática entre < y >, representan constructos de la misma naturaleza con diferentes implementaciones. Son los siguientes:

- SECTION: Agrupa un conjunto de capas o bloques destinados a realizar funciones específicas. Las secciones organizan y segmentan la red neuronal en componentes manejables, permitiendo una mejor gestión y optimización del flujo de datos a través de la red.
- LAYER: Son los componentes fundamentales de los bloques en una red neuronal, que realizan funciones específicas en el procesamiento de datos.
- OPTIMIZER: Algoritmo utilizado para ajustar los pesos de la red neuronal durante el proceso de entrenamiento, mejorando su precisión.
- LOSS-FUNCTION: Función que mide la discrepancia entre las predicciones de la red y los valores reales, guiando el proceso de optimización para minimizar el error.
- STRATEGY: Técnicas específicas utilizadas en el entrenamiento de la red neuronal para mejorar su rendimiento y eficacia.
- EARLY-STOPPER: Mecanismo que detiene el entrenamiento de la red cuando no se observa una mejora significativa en un conjunto de datos de validación.
- CHECK-POINT-SAVER: Herramienta que guarda el estado actual de la red en intervalos regulares durante el entrenamiento, permitiendo recuperar el modelo en caso de interrupciones.
- SHORTCUT: Permite la conexión directa entre la entrada y la salida de un bloque, facilitando el flujo de información y ayudando a evitar el problema de la desaparición del gradiente en redes profundas.
- FACET: Mecanismo que permite modificar y personalizar el comportamiento de un bloque en una red neuronal, ajustándolo a necesidades específicas.

En el anexo B figuran los constructos que se corresponden con estas palabras claves extensibles.

3.2.2. Sintaxis

La sintaxis, que se muestra en las figuras 3.4 y 3.5, da soporte a la definición de bloques para representar los constructos de arquitectura y laboratorio.

```
Architecture ::= 'Architecture' NAME Constructor? NL INDENT Properties* Section+ DEDENT
Section ::= <SECTION> Constructor? NL INDENT Properties* Input? Block+ DEDENT
Block ::= 'Block' Constructor? Facet? NL INDENT Properties* Shortcut? Layer+ DEDENT
Shortcut ::= <SHORTCUT> Constructor? NL INDENT Properties* Layer+ DEDENT
Layer ::= <LAYER> Constructor? (NL INDENT Properties* Output? DEDENT)?
Constructor ::= '(' Property (',' Property)* ')'
Properties ::= Property NL
Property ::= NAME '=' VALUE
Facet ::= 'as' <FACET> Constructor?
```

Ilustración 3.4: Sintaxis para la definición de la arquitectura en EBNF

```
Laboratory ::= 'Laboratory' Constructor? NL INDENT Properties* LabDefinition Experiment+ DEDENT
LabDefinition ::= Optimizer LossFunction Dataset Strategy EarlyStopper? CheckPointSaver?
Experiment ::= 'EXPERIMENT' Constructor? (NL INDENT Properties* ExperimentDefinition DEDENT)?
ExperimentDefinition ::= Optimizer? LossFunction? Materialization*
Materialization ::= 'Materialization' Constructor? NL INDENT Properties* Layer DEDENT
Optimizer ::= <OPTIMIZER> Constructor (NL INDENT Properties* DEDENT)?
LossFunction ::= <LOSS-FUNCTION> Constructor (NL INDENT Properties* DEDENT)?
Dataset ::= 'Dataset' NL INDENT Properties* Split DEDENT
Strategy ::= <STRATEGY> Constructor? (NL INDENT Properties* DEDENT)?
EarlyStopper ::= <EARLY-STOPPER> Constructor? (NL INDENT Properties* DEDENT)?
CheckPointSaver ::= <CHECK-POINT-SAVER> Constructor? (NL INDENT Properties* DEDENT)?
Split ::= 'Split' Constructor (NL INDENT Properties* DEDENT)?
```

Ilustración 3.5: Sintaxis para la definición del laboratorio en EBNF

En el *DSL*, cada bloque del lenguaje puede ser configurada con propiedades. La definición de estas propiedades no está fijada en la gramática, sino se controla mediante reglas semánticas como veremos en la sección 3.2.3. Esto permite que el lenguaje se pueda ajustar flexiblemente sin necesidad de modificar la gramática. Las propiedades pueden ser declaradas de dos maneras:

- Dentro del constructor. Donde se presentan separadas por comas y entre paréntesis. Este método es útil para definir configuraciones esenciales para la creación del componente.
- En el cuerpo del bloque. Donde se presentan separadas por saltos de línea. Este método permite una declaración más extensa y detallada de las propiedades, proporcionando un espacio para configuraciones adicionales.

La arquitectura representa el esqueleto de la red neuronal. Se inicia con un identificador que define el nombre de la arquitectura, seguido opcionalmente de un constructor. En el cuerpo se declaran las secciones que componen la red. Una Sección dentro sirve para dividir

la red en partes lógicamente coherentes. Cada sección puede contener definiciones específicas de cómo los datos son transformados a través de bloques para procesamiento de características, detección de patrones, etc. La sección puede incluir opcionalmente una declaración del tamaño del input. Un bloque es una unidad funcional dentro de una sección que define una agrupación de capas. En algunos caso es necesario declarar *Shorcuts* para conectar la entrada con la salida. Por último, la capa es el componente más granular en la definición de una arquitectura. Representa una unidad de procesamiento que realiza transformaciones específicas sobre los datos de entrada para producir una salida. Cada capa puede tener configuraciones particulares que determinan su funcionamiento, tales como el tipo de operaciones que realiza, los parámetros que utiliza, y cómo contribuye al objetivo general de la sección o la arquitectura en su totalidad. No obstante, también se pueden definir capas Virtuales, que se materializan en la definición de los experimentos del laboratorio. La posibilidad de definir capas virtuales añade una dimensión de parametrización a la arquitectura de las redes neuronales. Esto significa que una arquitectura no necesita ser un diseño rígido, sino que puede ser concebida como una familia de arquitecturas, donde distintos parámetros pueden ser ajustados para explorar variaciones en el diseño. Esto es particularmente útil en la investigación y desarrollo de nuevos modelos donde se desea evaluar el impacto de diferentes configuraciones estructurales sin necesidad de redefinir completamente la arquitectura básica.

Por otro lado, el laboratorio se inicia con un identificador único, que le da nombre al entorno de entrenamiento, seguido opcionalmente de un constructor que puede contener propiedades esenciales, tales como el numero de iteraciones o el nombre del mismo. Dentro del cuerpo del laboratorio, se declaran experimentos y la configuración del entrenamiento, que incluye:

- **Optimizer** A.1. Define como se ajusten los pesos de la red con el objetivo de minimizar la función de pérdida. Se incluyen SGD (*Stochastic Gradient Descent*), *Adam*, *RMSprop*, entre otros.
- **LossFunction** A.2. Define como se mide el error la respuesta del modelo con respecto al valor esperado. Se incluyen la entropía cruzada para clasificación y el error cuadrático medio para regresión, entre otros.
- **Dataset**. Es el conjunto de datos utilizado para entrenar, validar y probar el modelo. Con el component Split se dividen los datos en un conjunto de entrenamiento, uno de validación y otro de prueba.
- **Strategy**. Se define cómo se entrena el modelo en términos de velocidad y eficacia, y cómo se maneja la complejidad computacional y la capacidad de generalización del modelo.
- **CheckPointSaver**. Es también un componente opcional que se encarga de guardar el estado del modelo en intervalos específicos durante el entrenamiento. Esto es útil para recuperar y reanudar el entrenamiento desde un punto guardado en caso de que el proceso sea interrumpido. Además, permite seleccionar la versión del modelo que haya demostrado el mejor rendimiento según algún criterio definido.
- **EarlyStopper**. Es un componente opcional que permite detener el entrenamiento antes de completar todas las iteraciones previstas si se cumplen ciertas condiciones, como

cuando el error de validación deja de mejorar. Este mecanismo ayuda a prevenir el sobreajuste y puede mejorar la eficiencia del entrenamiento al evitar cálculos innecesarios.

Los experimentos están diseñados para probar y validar diferentes configuraciones y parámetros de la red. Cada experimento puede cambiar el *Optimizer* y la *Loss Function*, pero lo más relevante es que permite especificar cómo se materializan las capas virtuales de la arquitectura del modelo.

3.2.3. Semántica

La semántica del *DSL* de *Flogo* dicta un conjunto de reglas que aseguran la validez y la coherencia del modelo definido dentro del *DSL*. Estas reglas no solo especifican el significado de cada componente y constructo sintáctico, sino que también validan su implementación correcta y lógica dentro del contexto del modelo.

En cualquier modelo definido en el *DSL* de *Flogo*, sólo puede haber una instancia de arquitectura y una de laboratorio. No es posible definir múltiples laboratorios para una arquitectura. En su lugar, sería más apropiado establecer varios experimentos dentro de un único laboratorio para explorar diversas configuraciones de entrenamiento. Del mismo modo, no se permite la creación de múltiples arquitecturas para un solo laboratorio. Lo recomendado sería diseñar una arquitectura que sea parametrizable, ajustando sus parámetros a través de los experimentos definidos en el laboratorio a través de las *VLayers*.

En el laboratorio se pueden definir varios experimentos. Cada experimento puede incluir opcionalmente sus propias funciones de pérdida y optimizadores, pero el laboratorio en su conjunto debe definir obligatoriamente un optimizador, una función de pérdida y un dataset. Esta estructura asegura que el modelo tenga todas las herramientas necesarias para el entrenamiento y evaluación.

La arquitectura debe contener una o más secciones. Cada sección puede albergar múltiples bloques, y cada bloque puede contener una o más capas. Esta estructura jerárquica facilita la legibilidad de los modelos de redes neuronales.

En la parametrización de las *convolutional layers* B.1 y de las *pool layers* B.1 se puede realizar de dos maneras distintas. Definir la capa utilizando el *Kernel* y el número de canales de salida. O Alternativamente, se puede utilizar el constructo *Output* para determinar directamente el tamaño de salida deseado de la capa. Esta flexibilidad permite a los programadores ajustar la capa a necesidades sus necesidades.

Dentro de una sección recurrente en el *DSL* de *Flogo*, las capas recurrentes B.1 deben ser definidas exclusivamente al comienzo de la sección. Esto se debe a que estas capas son cruciales para manejar los datos que se procesan de forma secuencial y tienen la responsabilidad de transformar la información de dos dimensiones a una dimensión única. Esta transformación se realiza mediante un *Map-Reduce* configurable.

Capítulo 4

Implementación

En este capítulo abordamos la estrategia de implementación del *DSL*. Hay muchas alternativas para el desarrollo de lenguajes específicos de dominio (*DSL*), cada una con sus ventajas y problemas. Una estrategia es la **implementación directa**, que implica la creación del *DSL* desde cero, desarrollando todos los componentes necesarios, incluidos el analizador léxico, el analizador sintáctico y el generador de código. Aunque esta aproximación proporciona el mayor nivel de control sobre el lenguaje y su comportamiento, también es la más costosa y compleja, requiriendo un profundo conocimiento en teoría de lenguajes, técnicas de análisis y generación de código.

Otra estrategia es el uso de herramientas especializadas que ayudan a reducir significativamente el esfuerzo necesario. Algunas de las herramientas más destacadas en esta categoría incluyen:

- **ANTLR** (*ANother Tool for Language Recognition*) [31]. Genera analizadores léxicos y sintácticos a partir de gramáticas escritas en un formato específico, simplificando la creación de lenguajes personalizados.
- **JetBrains MPS**. Una plataforma que permite a los desarrolladores crear y utilizar *DSLs* para resolver problemas específicos dentro de un dominio [7].
- **Xtext**. Una herramienta de metaprogramación basada en Eclipse que se utiliza para desarrollar *DSLs* en Java, ofreciendo alta flexibilidad y personalización [3].

Para desarrollar *Flogo*, hemos optado por un método llamado Intino, desarrollado por el Instituto Universitario SIANI, basado en *Model Driven Engineering* (MDE). Este enfoque nos permite estructurar y gestionar el desarrollo de *DSLs* a través de dos niveles de abstracción: entidades y conceptos.

- **Entidades**: Representan los elementos más concretos y específicos dentro del dominio. En este nivel, se modelan los objetos individuales con todas sus propiedades y relaciones detalladas.
- **Conceptos**: Agrupan entidades que comparten características comunes, proporcionando una vista más abstracta y generalizada del dominio. Los conceptos permiten definir las propiedades y comportamientos que deben tener todas las entidades de una misma

categoría.

En el modelo se representan las entidades concretas. Por ejemplo, en el contexto de *Flogo*, un modelo podría incluir elementos específicos una red neuronal concreta. Por otro lado, en el metamodelo se representan los conceptos, que definen la estructura y reglas que deben seguir los modelos. Por ejemplo, el metamodelo de *Flogo* incluye todos los constructos descritos en el capítulo 3, que constituyen el marco analítico con el que el programador se enfrenta a la tarea de describir una red neuronal.

Intino permite la construcción automática del código necesario mediante un proceso estructurado en cuatro pasos principales:

1. **Definición del metamodelo**. En esta fase, se definen los constructos necesarios para modelar redes neuronales. Esto incluye la especificación de las propiedades, relaciones y restricciones que deben seguir los modelos.
2. **Generación del DSL**. A partir del metamodelo, Intino genera automáticamente el *DSL*, garantizando que todas las reglas y estructuras definidas en el metamodelo se reflejen correctamente en el lenguaje. Este *DSL* incluye el léxico, la sintaxis y la semántica necesarios para modelar redes neuronales de manera efectiva.
3. **Definición del modelo**. Con el nuevo *DSL* generado, los programadores pueden implementar modelos específicos de redes neuronales, definiendo las entidades a partir de los constructos definidos en el metamodelo.
4. **Generación de código**. A partir del modelo de la red neuronal, hay que implementar un generador de código que se encarga de producir los dos programas necesarios: la arquitectura de la red neuronal y el proceso de entrenamiento.

La definición del metamodelo es la primera tarea y es fundamental, ya que implica la creación de los constructos necesarios para modelar redes neuronales. La generación de código es la cuarta tarea y consiste en generar automáticamente dos programas esenciales para obtener un modelo de la red neuronal: la arquitectura de la red neuronal y el proceso de entrenamiento. Estas dos tareas se desarrollarán en detalle en las secciones 4.1 y 4.2 de este capítulo.

La segunda y tercera tarea no requieren un desarrollo. Por un lado, la generación del *DSL*, se realiza automáticamente por la herramienta de Intino. Este proceso incluyen la generación del léxico, la sintaxis y la semántica necesarios para modelar redes neuronales de manera efectiva. La definición del modelo, es responsabilidad del desarrollador que quiera crear un modelo de inteligencia artificial. Utilizando el *DSL* generado, los programadores podrán implementar modelos específicos de redes neuronales.

4.1. Definición del metamodelo

Como se ha dicho, el desarrollo del *DSL* requiere definir el metamodelo. Para ello hemos usado un *DSL* de Intino llamado Proteo, especializado en el modelado de conceptos. En la figura 4.1 vemos una parte del metamodelo, relacionado con los constructos para la

configuración y ejecución de experimentos en un laboratorio.

```
dsl Proteo

Concept:{1..1} Laboratory
    var integer eras=1
    var integer epochs
    var string name
    var word:{CPU GPU MPS Default} device=Default
    has:{1..1} Optimizer
    has:{1..1} LossFunction
    has:{0..1} EarlyStopper
    has:{1..1} Strategy
    has:{0..1} CheckPointSaver
    Concept:{1..1} Dataset
        var string name
        var integer batchSize=10
        Concept:{1..1} Split
            var double train
            var double test
            var double validation
        Concept:{1..*} Experiment
            has:{0..1} Optimizer
            has:{0..1} LossFunction
```

Ilustración 4.1: Metamodelo del laboratorio en *Proteo*

Al crear un metamodelo, es fundamental comprender y definir claramente los conceptos y elementos esenciales que conforman el sistema. En este contexto, se identifican tres estructuras clave: la definición de constructos, la definición de atributos y la definición de componentes.

Los constructos son las entidades abstractas o conceptos principales que el metamodelo debe representar. Permite la creación de un marco conceptual abstracto que puede aplicarse a diferentes contextos y casos de uso.

La palabra clave `Concept` se utiliza para definir constructos como el de laboratorio o *Dataset*. Estos constructos se convierten en palabras clave del *DSL* de *Flogo* como se puede ver en la figura 4.2.

Los atributos son las propiedades o características que describen y diferencian a los constructos. Cada constructo puede tener uno o más atributos que especifican información relevante sobre dicho constructo. La definición de atributos implica identificar qué propiedades son necesarias para describir adecuadamente cada constructo dentro del metamodelo. Proporciona un nivel de detalle necesario para la correcta representación y manipulación de los constructos, y ayuda a distinguir entre diferentes instancias de un mismo constructo, facilitando la gestión y el análisis de datos.

La palabra clave `var` se utiliza para definir variables dentro de los conceptos. Estas variables pueden ser de diferentes tipos, como integer, string, word (para valores enumerados) y double.

```
dsl Flogo

Laboratory(epochs=10, name="MNIST EXAMPLE")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name="mnist_flatten.tsv",batchSize=10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626

    Experiment e221
        MSELoss
```

Ilustración 4.2: Modelo con la definición de un laboratorio en *Flogo*

En el ejemplo, `eras` es un atributo entero con un valor predeterminado de 1, utilizado para indicar el número de iteraciones o ciclos que se realizarán en un proceso. Cuando se modela un laboratorio, es opcional indicar el valor de este atributo. Por el contrario, `epochs` y `name` son variables esenciales: `epochs`, una variable entera que representa el número total de iteraciones de entrenamiento, y `name`, una variable de cadena que define el nombre del modelo o del experimento. Estos no tienen valores predeterminados, por lo que es obligatorio definirlos en el modelado del laboratorio. `device` es un atributo de tipo palabra (enumeración) con posibles valores `CPU`, `GPU`, `MPS` y `Default`, siendo `Default` el valor predeterminado. Este atributo especifica el tipo de hardware en el que se ejecutará el modelo, permitiendo una optimización adecuada según el dispositivo disponible.

Los componentes son las partes o unidades funcionales del sistema que interactúan para formar el sistema completo. En el contexto de un metamodelo, los componentes pueden ser entidades físicas o lógicas que tienen un rol específico dentro del sistema. La definición de componentes incluye la identificación de estos elementos y la descripción de cómo se integran e interactúan dentro del metamodelo.

La palabra clave `has` se utiliza para definir componentes dentro de los conceptos, especificando las relaciones y cardinalidades de estos componentes. Estos componentes son, a su vez, conceptos que están descritos en otra parte del metamodelo. Esto permite una estructura modular y detallada de los elementos y sus interrelaciones en el modelado del laboratorio.

En el ejemplo, `Optimizer`, `LossFunction` y `Strategy` son componentes obligatorios, mientras que `EarlyStopper` y `CheckPointSaver` son opcionales, tal y como se define por su cardinalidad. La cardinalidad se especifica utilizando el formato {min..max}, indicando el número mínimo y máximo de instancias permitidas para una variable o componente. En el ejemplo, Laboratory y Dataset deben tener exactamente una instancia (1..1), mientras que Experiment puede tener una o más instancias (1..*). Estos componentes tienen funciones específicas para el entrenamiento del modelo:

- **Optimizer**. Se utiliza para ajustar los parámetros del modelo con el fin de minimizar la función de pérdida.
- **LossFunction**. Define cómo se calcula el error entre las predicciones del modelo y los

valores reales, lo cual es crucial para el proceso de optimización.

- **Strategy**. Establece el enfoque o método que se utilizará para resolver un problema específico dentro del modelo.
- **EarlyStopper**. Monitorea el entrenamiento del modelo y lo detiene cuando las mejoras en la métrica de rendimiento son insignificantes, previniendo el sobreajuste.
- **CheckPointSaver**. Guarda periódicamente el estado del modelo durante el entrenamiento, permitiendo recuperarlo en caso de interrupciones o para analizar versiones anteriores.

4.2. Generador de código

La generación de código en el contexto de la Ingeniería Dirigida por Modelos (MDE) se centra en automatizar la creación de aplicaciones a partir de modelos de alto nivel. Esto reduce el tiempo dedicado a la generación manual de código y por lo tanto se disminuye la probabilidad de errores. Esto es especialmente importante en proyectos de gran escala donde la consistencia y la precisión del código son críticas. En el caso de *Flogo*, se generan dos aplicaciones: la arquitectura de una red neuronal y el programa que automatiza el entrenamiento del modelo.

La figura 4.3 ilustra la arquitectura del generador de código. Dentro de esta arquitectura, algunos elementos son proporcionados por Intino (con fondo verde), mientras que otros son ofrecidos por *ItRules*[32] (con fondo azul), un sistema de generación de código basado en plantillas. Los motores de plantillas son fundamentales en la generación automática de código. A continuación, se detalla la función de cada componente y su contribución al funcionamiento global del sistema.

En primer lugar, Intino ofrece componentes clave que facilitan el procesamiento y la interpretación del código fuente o de la entrada proporcionada. Estos componentes son:

- DSL Parser. Este es el primer punto de contacto en el flujo de procesamiento del sistema. Su función principal es analizar y comprender la estructura del metamodelo.
- *Syntactic Graph*, que representa la estructura del metamodelo.

Por otro lado, *ItRules* proporciona el motor de plantillas (*Template Engine*) que es responsable de la generación dinámica de código basado en plantillas predefinidas.

Además, el generador cuenta con otros módulos importantes. Por un lado el *Model Mapper*, que se encarga de mapear el grafo sintáctico al modelo de la vista (*View Model*). Esta transformación se realiza para que el motor de plantillas tenga los datos del modelo organizados conforme lo requieren las plantillas de código. Por otro lado, el generador de código (*Code Generator*). Este componente coordina la ejecución del proceso, tomando los modelos mapeados para automatizar la producción de código.

Es decir, a partir del modelo de la red neuronal, especificada con el *DSL* de *Flogo*, el analizador sintáctico de Intino genera un grafo sintáctico con nodos y relaciones que representan

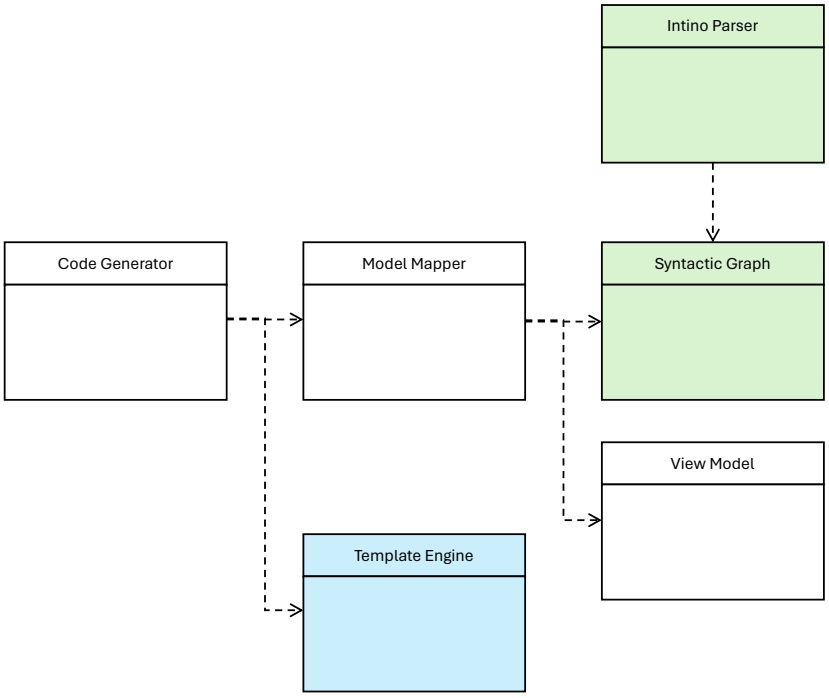

Ilustración 4.3: Diagrama UML del generador de código de Flogo

dicha red neuronal. Este grafo es una representación estructurada de la red, donde los nodos representan elementos como capas y bloques, y las relaciones entre los nodos indican cómo estos elementos están conectados. No obstante, el grafo sintáctico generado por el parser no es directamente válido para la generación de código porque su estructura no coincide con la esperada por las plantillas. Es aquí donde entra en juego la necesidad de transformar el grafo sintáctico en una vista compatible con las plantillas que se utilizan para generar el código.

Este proceso de transformación asegura que los datos del grafo sintáctico se ajusten a la estructura requerida por las plantillas, cumpliendo con el contrato definido con los *frameworks* estructural y operacional. En este contexto, se usan dos conjuntos específicos de plantillas, definidas por el *framework* estructural y el *framework* operacional.

- El *framework* estructural se encarga de dar soporte a la arquitectura de la red neuronal. A partir de las plantillas del *framework* estructural, el generador de código crea el código mediante clases de este *framework* que representan las capas, secciones y bloques de la red.
- El *framework* operacional se ocupa de dar soporte a los scripts que automatizan el entrenamiento del modelo, configurando parámetros clave como conjuntos de datos, métricas de evaluación y métodos de optimización.

Estos *frameworks* son el resultado de dos trabajos de fin de título realizados por Juan Carlos Santana Santana y Joel del Rosario Pérez. La integración entre los proyectos consistió por un lado en asegurar que el generador de código pudiera utilizar de manera efectiva las plantillas definidas por ambos *frameworks*, y por otro lado asegurar que las salidas generadas por el *framework* estructural sean compatibles y fácilmente integrables con los procesos

definidos por el *framework* operacional.

4.2.1. Integración con el *framework* estructural

El generador de código utiliza las plantillas asociadas al *framework* estructural para generar el código que implementa dicha estructura. Esto incluye la creación de clases y funciones que representan las capas, neuronas y conexiones de la red neuronal.

En la figura 4.4 ilustra la relación entre la arquitectura de la red neuronal y el *framework* estructural. El *framework* estructural proporciona una interfaz y un conjunto de clases base que facilitan la implementación y extensión de la funcionalidad. Al ser orientado a objetos, permite que las nuevas clases derivadas extiendan las clases del *framework*, promoviendo la modularidad. Además, el *framework* estructural actúa como una capa de recubrimiento que aísla del uso de herramientas como *PyTorch*. Esto significa que extendiendo el *framework* estructural se pueden hacer redes neuronales sin necesidad de interactuar directamente con los componentes más complejos de *PyTorch*. La arquitectura de la red neuronal contiene al código que define la red neuronal. Incluye la especificación de las capas y conexiones entre ellas y otros parámetros críticos para el funcionamiento de la red.

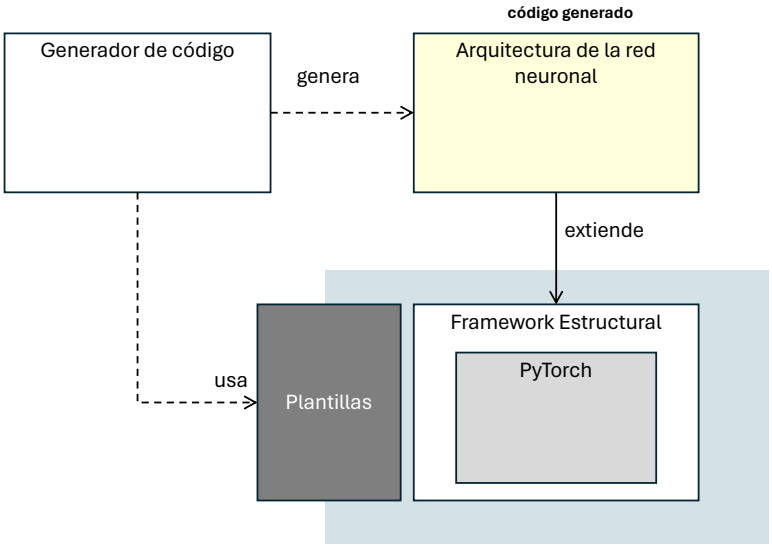

Ilustración 4.4: Integración con el *framework* estructural

El código de la arquitectura de la red neuronal es generado automáticamente por el generador de código usando las plantillas que proporciona el *framework* estructural. En la figura 4.5 se muestra un ejemplo de una arquitectura simple modelada con el *DSL* de *Flogo* y el código generado para dicho *framework*.

```
    Experiment e626
        Materialization(vLayer="01") > Linear > Output(x=300)
        Materialization(vLayer="ActivationLayer") > ReLU

    Experiment e501
        Materialization(vLayer="01") > Linear > Output(x=500)
        Materialization(vLayer="ActivationLayer") > Sigmoid


Architecture
    RecurrentSection
        Input(seqLength = 28, x = 28)
        Block
            LSTM(bidirectional = false, numLayers = 4)
                EndSequence(reduce = flatten) > Output(x=512)
            VLayer(id="ActivationLayer")
        Block
            VLayer(id="01")
            VLayer(id="ActivationLayer")
        Block
            Linear > Output(x=75)
            VLayer(id="ActivationLayer")
        Block
            Linear > Output(x=10)
            Softmax
```

```
architecture626 = (Architecture("e626").attach(recurrentSection([
  Block([LSTMLayer(input_size=28, hidden_size=512,
                output_type=LSTMLayer.OutputType.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)
  ])])))

architecture501 = (Architecture("501").attach(recurrentSection([
  Block([LSTMLayer(input_size=28, hidden_size=512,
                output_type=LSTMLayer.OutputType.EndSequence,
                num_layer=4, bidirectional=False, dropout=0.0),
      FlattenLayer(from_dim=2, to_dim=1), SigmoidLayer()]),
  Block([LinearLayer(in_features=14336, out_features=500,
                 dimension=-1, bias=True), SigmoidLayer()]),
  Block([LinearLayer(in_features=500, out_features=75,
                 dimension=-1, bias=True), SigmoidLayer()]),
  Block([LinearLayer(in_features=75, out_features=10,
                 dimension=-1, bias=True), SoftmaxLayer(dimension=-1)
  ])])))
```

Ilustración 4.5: Ejemplo de generación de código estructural

Como se puede ver, el código generado no tiene ninguna dependencia con *PyTorch* lo que eventualmente permitiría reemplazar esta librería por otra, o actualizarla a una versión más reciente, sin hacer ningún cambio en el código desarrollado.

4.2.2. Integración con el *framework* operacional

El *framework* operacional proporciona el soporte para automatizar el proceso de entrenamiento del modelo. En la figura 4.6 ilustra los componentes principales y sus interacciones, describiendo cómo se coordina el entrenamiento de redes neuronales en *Flogo*.

El *framework* operacional proporciona un conjunto de clases base que soporta y automatiza el proceso de entrenamiento. Este *framework* interactúa, a través de una *API*, con un *web service* que permite que las solicitudes de entrenamiento sean manejadas de manera remota, en una infraestructura de entrenamiento. A través de la *API*, el *web service* puede iniciar, monitorizar y gestionar sesiones de entrenamiento, proporcionando una interfaz accesible tanto para usuarios y desarrolladores.

A partir de este *framework* se extiende el programa que permite el entrenamiento de la red neuronal. Aquí es donde se definen y ejecutan las iteraciones de entrenamiento, se ajustan los parámetros del modelo y se evalúa su rendimiento en función de los datos de entrenamiento.

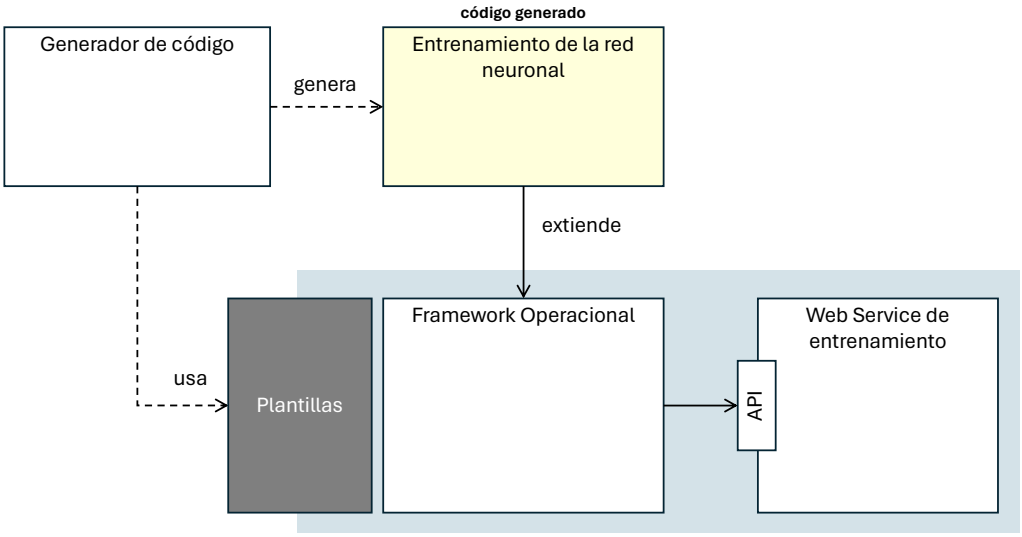

Ilustración 4.6: Integración con el *framework* operacional

La *API REST* desarrollada para gestionar los archivos de arquitectura, laboratorios generados por los usuarios de Flogo y los datasets que vayan a ser usados. Esta *API* permite ejecutar entrenamientos y visualizar resultados. Contiene las siguientes funciones:

- Añadir recurso. Permite subir arquitecturas, laboratorios y datasets. Por ejemplo, cuando se realiza una solicitud POST a la ruta /flogo/architecture, permite añadir una arquitectura al repositorio.
- Obtener recurso. Permite obtener recursos almacenados en el repositorio mediante una solicitud GET a una ruta. Por ejemplo, /flogo/architecture/:name, donde :name es el nombre de una arquitectura. La respuesta incluirá el recurso solicitado en el cuerpo de la respuesta.
- Listar recursos. Lista todos los recursos de un determinado tipo subidos al repositorio realizando una solicitud GET a una ruta. Por ejemplo, /flogo/architecture.
- Eliminar recurso. Permite eliminar recursos del repositorio mediante una solicitud DELETE a una ruta. Por ejemplo /flogo/architecture/:name, donde :name es el nombre del archivo. La respuesta confirmará si la eliminación fue exitosa.
- Ejecutar un laboratorio. Ejecuta el entrenamiento de un laboratorio especificado mediante una solicitud POST a la ruta /flogo/execute/:laboratory, donde :laboratory es el nombre del laboratorio. La respuesta será un objeto JSON que indica el nombre de la arquitectura que mejor rendimiento obtuvo.
- Obtención de estadísticas. Permite acceder a las estadísticas de los laboratorios ejecutados mediante una solicitud GET a la ruta /flogo/stats/:stat, donde :stat es el nombre de la estadística deseada. Los parámetros adicionales se añaden como query parameters. La respuesta incluirá las estadísticas solicitadas en formato JSON.
- Obtención del modelo entrenado. Proporciona una imagen Docker mediante una solicitud GET a la ruta /flogo/model/:experiment, donde :experiment es el nombre del

experimento.

El código de entrenamiento de la red neuronal es generado automáticamente por el generador de código usando las plantillas que proporciona el *framework* operacional. En la figura 4.7 se muestra un ejemplo de un entrenamiento simple modelado con el *DSL* de *Flogo* y el código generado para dicho *framework*.

```
dsl Flogo

Laboratory(epochs = 10, name = "MNIST")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name = "mnist.csv", batchSize = 10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626
        Materialization(vLayer="01") > Linear > Output(x=300)
        Materialization(vLayer="ActivationLayer") > ReLU

    Experiment e501
        Materialization(vLayer="01") > Linear > Output(x=500)
        Materialization(vLayer="ActivationLayer") > Sigmoid
```

```
dataset = DatasetGenerator(name=“mnist.
   batch_size=10, random_state=389)
 .generate(train_proportion=0.7. validation_proportion=0.1, test_proportion=0.2)

optimizer626 = AdamOptimizer(parameters=e626.parameters(),
  learning_rate=1.0E-4, betas=(0.9, 0.999),
  eps=1.0E-8, weight_decay=0.0)

optimizer501 = AdamOptimizer(parameters=e501.parameters(),
  learning_rate=1.0E-4, betas=(0.9, 0.999),
  eps=1.0E-8, weight_decay=0.0)

experiments = [Experiment(name=“e626", architecture=e626, optimizer= optimizer626,
  loss_function=MAELossFunction(), saver=ModelSaver("")),
               Experiment(name=“e501", architecture=e501, optimizer=optimizer501,
  loss_function=MAELossFunction(), saver=ModelSaver(""))]

Laboratory(name=“MNIST", eras=1, epochs=10, datagen=dataset,
  experiments=experiments,
  strategy=ClassificationStrategy(), logger=Logger(“"),
  loader=ModelLoader(), device=Device(-1))
.explore()
```

Ilustración 4.7: Ejemplo de generación de código operacional

Capítulo 5

Ejemplos

En este capítulo, nos enfocaremos en los usos del *DSL* (*Domain Specific Language*) de *Flogo*, una plataforma que permite construir aplicaciones de integración, procesamiento de eventos y funciones *Serverless* de manera eficiente. Exploraremos cómo *Flogo* puede ser aplicado en diferentes contextos para resolver problemas reales, destacando su flexibilidad y facilidad de uso. Para ilustrar estos conceptos, presentaremos varios ejemplos y casos de uso en los que *Flogo* se utiliza para manejar tareas de creación de modelos basados en redes neuronales.

La pragmática se refiere a cómo se aplica este lenguaje en situaciones prácticas del mundo real. En el contexto de la programación, la pragmática se puede definir como el uso efectivo y práctico de un lenguaje de programación para resolver problemas concretos. Esto no solo incluye cómo escribir el código, sino también cómo implementarlo para obtener resultados tangibles. Para demostrar cómo se puede aplicar de manera práctica el *DSL* de *Flogo*, proporcionamos varios ejemplos detallado para mostrar cómo este *DSL* puede ser utilizado en situaciones reales.

Como hilo conductor de los ejemplos que se presentan en este capítulo, utilizaremos como dataset de entrenamiento MNIST [9] (*Modified National Institute of Standards and Technology*). Este es un dataset ampliamente utilizado en el campo del aprendizaje automático. Contiene 70,000 imágenes de dígitos escritos a mano, divididas en un conjunto de entrenamiento de 60,000 imágenes y un conjunto de prueba de 10,000 imágenes. Cada imagen es de tamaño 28x28x1 píxeles y está etiquetada con el dígito que representa, que va del 0 al 9, tal y como se puede ver en la figura 5.1.

El MNIST se ha convertido en un estándar de referencia para evaluar y comparar el rendimiento de diferentes algoritmos de aprendizaje automático y modelos de redes neuronales. Su popularidad se debe a su simplicidad y a la disponibilidad de un gran número de muestras, lo que facilita el entrenamiento y la validación de modelos. Además, al ser un dataset relativamente pequeño en términos de complejidad, permite a los investigadores y desarrolladores realizar experimentos rápidos y efectivos. El MNIST es considerado a menudo como un *Toy Problem*, en el sentido de que es un problema simplificado y bien definido que permite a los investigadores probar y perfeccionar sus modelos en un entorno controlado.

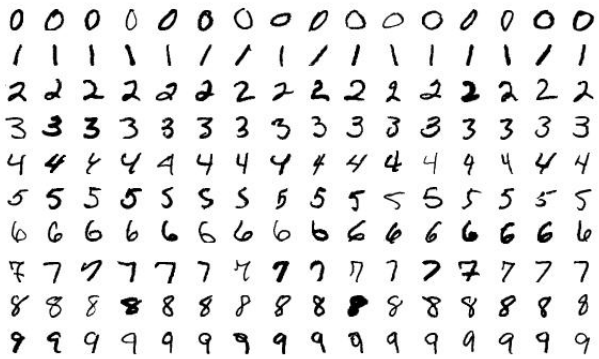

Ilustración 5.1: Dataset MNIST usado en los ejemplos de Flogo

5.1. Perceptrón multi-capa

En la figura 5.2 se muestra una arquitectura y un laboratorio de entrenamiento diseñados para una red neuronal modelada con el *DSL* de *Flogo*.

En primer lugar, en este ejemplo podemos ver que la arquitectura de la red neuronal es un perceptrón multi-capa. La arquitectura se espera como entrada un tensor de tamaño 784 (correspondiente a las 28x28 píxeles de las imágenes aplanadas de MNIST), y genera como salida un tensor de tamaño 10, que se corresponde con cada uno de los dígitos que tiene que reconocer. Los bloques son los siguientes:

- El primer bloque tiene una capa lineal con 200 neuronas de salida seguida de una función de activación ReLU.
- El segundo bloque también tiene una capa lineal, esta vez con 84 neuronas de salida, seguida de una función de activación ReLU.
- El tercer y último bloque tiene una capa lineal con 10 neuronas de salida, seguida de una función de activación Softmax, la cual es adecuada para tareas de clasificación con 10 clases (dígitos del 0 al 9).

En segundo lugar, se modela también un laboratorio donde especificamos:

- El optimizador, que se utiliza Adam
- La función de pérdida, que en este caso es MAELoss (Mean Absolute Error Loss)
- La estrategia de entrenamiento, que en este caso es clasificación
- El dataset con el que vamos a entrenar el modelo, que en este caso es el dataset MNIST.

El dataset se carga desde un archivo denominado “mnist_flatten.tsvz se procesa en lotes de tamaño 10. La división de los datos se realiza de la siguiente manera: entrenamiento 70 %, prueba 20 %, y validación 10 %. Además, definimos un experimento, en este caso denominado `e626`, cuya función es encapsular la tarea de entrenamiento.

```
Laboratory(epochs = 10, name = "MNIST EXAMPLE")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name = "mnist_flatten.tsv", batchSize = 10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626

Architecture LinearNeuralNetwork
    LinearSection
        Input(x=784)
        Block
            Linear > Output(x=200)
            ReLU
        Block
            Linear > Output(x=84)
            ReLU
        Block
            Linear > Output(x=10)
            Softmax
```

Ilustración 5.2: Perceptrón multi-capa definido en *Flogo*.

Se puede observar en este ejemplo que el modelo de la red neuronal descrita en *Flogo* puede ser transmitido de una manera sencilla, ya que es prácticamente una transcripción directa del lenguaje natural. Esto facilita la comprensión y la implementación, incluso para aquellos que no están familiarizados con detalles técnicos complejos. Además, es muy sencillo realizar el entrenamiento porque simplemente se especifican aquellas configuraciones que realmente aportan valor, eliminando la complejidad accidental. En el caso de que, después de entrenar esta arquitectura, no tuviéramos buenos resultados, el *DSL* nos permitiría iterar rápidamente. Por ejemplo, si creemos que debido a la función de pérdidas, en vez de modificar el código, añadiríamos un nuevo experimento como el que se ve en la figura 5.3. En este ejemplo, hemos añadido un nuevo experimento, llamado `e221`, dejando el experimento anterior para dejar constancia de que ya tuvimos un experimento previo.

```
Laboratory(epochs = 10, name = "MNIST EXAMPLE")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name = "mnist_flatten.tsv", batchSize = 10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626

    Experiment e221
       MSELoss

Architecture LinearNeuralNetwork
    LinearSection
        Input(x=784)
        Block
            Linear > Output(x=200)
            ReLU
        Block
            Linear > Output(x=84)
            ReLU
        Block
            Linear > Output(x=10)
            Softmax
```

Ilustración 5.3: Percentrón multi-capa definido en *Flogo* con dos experimentos.

Esta capacidad de *Flogo* nos permite tener trazabilidad de todos los experimentos que se van realizando. Lo que se suele hacer en otros *frameworks* es modificar el código, perdiendo la traza de lo que se había realizado anteriormente. Sin embargo, en *Flogo*, no es necesario realizar estos cambios; simplemente podemos añadir un nuevo experimento donde se utilice otra función de pérdidas. Esto no significa que haya que realizar los entrenamientos de todos los experimentos, el *framework* operacional no repite experimentos ya realizados. En la figura 5.4 vemos la misma arquitectura con varios experimentos, en los que especificamos Del mismo modo, otro optimizador o una combinación diferente de optimizador función de pérdida.

```
Laboratory(epochs = 10, name = "MNIST EXAMPLE")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name = "mnist_flatten.tsv", batchSize = 10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626

    Experiment e221
       MSELoss

    Experiment e500
       Adagrad

    Experiment e501
       Adagrad
       MSELoss

Architecture LinearNeuralNetwork
    LinearSection
        Input(x=784)
        Block
            Linear > Output(x=200)
            ReLU
        Block
            Linear > Output(x=84)
            ReLU
        Block
            Linear > Output(x=10)
            Softmax
```

Ilustración 5.4: Perceptrón multi-capa definido con el *DSL* de *Flogo* con cuatro experimentos.

En la figura 5.5, vemos como introducir una capa virtual (*VLayer*). Con esta capa, hemos parametrizado la arquitectura de forma que en los experimentos podremos indicar que capa de activación concreta queremos entrenar. Esto evita tener varias arquitecturas con una estructura muy parecida, lo que permite ajustar y probar diferentes configuraciones arquitectónicas de manera eficiente y sin complicaciones. Las *VLayers* promueven una mayor agilidad en el diseño y la implementación de modelos. Esta capacidad de parametrización se puede aplicar a cualquier capa de la red, tanto las de procesamiento como las de activación, ofreciendo una gran flexibilidad y control sobre la arquitectura del modelo.

```
Laboratory(epochs = 10, name = "MNIST EXAMPLE")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name = "mnist_flatten.tsv", batchSize = 10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626
       Materialization(vLayer="01") > ReLU

    Experiment e221
       MSELoss
       Materialization(vLayer="01") > Sigmoid

Architecture LinearNeuralNetwork
    LinearSection
        Input(x=784)
        Block
            Linear > Output(x=200)
            VLayer(id="01")
        Block
            Linear > Output(x=84)
            VLayer(id="01")
        Block
            Linear > Output(x=10)
            Softmax
```

Ilustración 5.5: Perceptrón multi-capa con capas virtuales.

Además, el DSL de *Flogo* permite especificar si se desea una conexión residual en un bloque. Al definir esta conexión, se podrá especificar las capas que deben ser incluidas en este *Shortcut* o si se desea se podrá definir una por defecto, proporcionando así una mayor flexibilidad en el diseño de la red.

5.2. Red neuronal convolucional

Exploraremos la configuración y el entrenamiento de una red neuronal convolucional aplicada al dataset MNIST. Utilizando el *DSL* de *Flogo*, mostraremos cómo se puede estructurar y entrenar esta red.

En primer lugar, en este ejemplo podemos ver que la arquitectura de la red neuronal de la figura 5.6 es una red neuronal convolucional. La arquitectura de la red neuronal es una red neuronal convolucional que toma como entrada un tensor de dimensiones 32x32x1 (correspondiente a imágenes en escala de grises de 32x32 píxeles), y genera como salida un tensor de tamaño 10, que se corresponde con cada una de las clases que tiene que reconocer (dígitos del 0 al 9). Las secciones son las siguientes:

- Sección Convolucional:
  - El primer bloque tiene una capa convolucional con 6 filtros que generan una salida de 28×28×6, seguida de una función de activación ReLU. A continuación, hay una capa de MaxPool que reduce las dimensiones a 14×14x6.
  - El segundo bloque tiene una capa convolucional con 16 filtros que generan una salida de 10×10×16, seguida de una función de activación ReLU. A continuación, hay una capa de MaxPool que reduce las dimensiones a 5×5x6.
- Sección de Aplanado:
  - Convierte el tensor 3D en un vector 1D para ser procesado por las capas de la siguiente sección.
- Sección Lineal:
  - El primer bloque tiene una capa lineal con 120 neuronas de salida seguida de una función de activación ReLU.
  - El segundo bloque también tiene una capa lineal, esta vez con 84 neuronas de salida, seguida de una función de activación ReLU.
  - El tercer y último bloque tiene una capa lineal con 10 neuronas de salida, seguida de una función de activación Softmax, la cual es adecuada para tareas de clasificación con 10 clases (dígitos del 0 al 9).

En segundo lugar, se modela también un laboratorio donde especificamos:

- El optimizador, que se utiliza es Adam.
- La función de pérdida, empleamos la MAELoss (*Mean Absolute Error Loss*).
- La estrategia de entrenamiento, es la de clasificación.
- El dataset con el que vamos a entrenar el modelo, en este caso es el dataset MNIST.

El dataset se carga desde un archivo denominado "mnist.tsv" y se procesa en lotes de tamaño 10. La división de los datos se realiza de la siguiente manera: entrenamiento 70 %, prueba 20 %, y validación 10 %. Además, definimos un experimento, el `e626`.

```
Laboratory(epochs = 10, name = "MNIST EXAMPLE")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name = “mnist.tsv", batchSize = 10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626

Architecture ConvolutionalNeuralNetwork
    ConvolutionalSection
        Input(32, 32, 1)
        Block
            Convolutional > Output(28, 28, 6)
            ReLU
            MaxPool > Output(14, 14)
        Block
            Convolutional > Output(10, 10, 16)
            ReLU
            MaxPool > Output(5, 5)

    FlattenSection

    LinearSection
        Block
            Linear > Output(120)
            ReLU
        Block
            Linear > Output(84)
            ReLU
       Block
            Linear > Output(10)
            Softmax
```

Ilustración 5.6: Red neuronal convolucional definida con el DSL de *Flogo*.

El constructo de *Output* en la configuración de las capas convolucional en el DSL de *Flogo* añade un nivel adicional de expresividad y flexibilidad. En *Flogo*, el uso de este constructo permite especificar claramente las dimensiones de salida de cada capa, lo cual es fundamental para entender cómo fluye la información a través de la red. Aunque *Flogo* también ofrece la posibilidad de detallar aún más los parámetros de la red tal y como podemos ver en la figura 5.7, se puede especificar tanto el tamaño del *kernel*, el *padding*, el *stride* como el número de canales de salida.

```
Convolutional
    Kernel
        Size(3, 3)
        Stride(2, 2)
        Padding(0, 0)
    OutChannels(6)
```

```
MaxPool
    Kernel
        Size(2, 2)
        Stride(2, 2)
        Padding(0, 0)
```

Ilustración 5.7: Configuración de las capas mediante el *Kernel* y el número de canales de salida.

Además, *Flogo* permite especificar si se desea una conexión residual en un bloque. Al definir esta conexión, se podrá especificar las capas que deben ser incluidas en este *Shortcut* o si se desea se podrá definir una por defecto, proporcionando así una mayor flexibilidad en el diseño de la red. Las conexiones residuales, populares en arquitecturas como *ResNet*, ayudan a mitigar problemas como el desvanecimiento del gradiente en redes profundas y facilitan el entrenamiento de modelos más complejos.

5.3. Red neuronal recurrente

En esta sección, exploraremos la configuración y el entrenamiento de una red neuronal recurrente (*RNN*) aplicada al conjunto de datos MNIST. Utilizando el *DSL* de *Flogo*, demostraremos cómo se puede estructurar y entrenar esta red de manera efectiva.

La arquitectura de la red neuronal que podemos ver en la ilustración 5.8 es una red neuronal recurrente que toma como entrada secuencias de longitud 28 y de tamaño 28 cada una (correspondiente a las filas de píxeles de las imágenes de 28x28 píxeles) y produce una salida de tamaño 10, que corresponde a las clases que debe reconocer (dígitos del 0 al 9). La estructura de la red se divide en varias secciones:

- Sección Recurrente:
  - La red recibe secuencias de longitud 28 con características de tamaño 28.
  - El primer bloque incluye una capa LSTM bidireccional con 4 capas, seguida de una operación de aplanado que genera una salida de tamaño 512.
  - Luego, se añade una capa lineal bidireccional con 4 capas, que produce una salida de tamaño 2000.
- Sección Lineal:
  - El primer bloque tiene una capa lineal con 500 neuronas, seguida de una función de activación ReLU.
  - El segundo bloque tiene una capa lineal con 100 neuronas, también seguida de una función de activación ReLU.
  - El tercer y último bloque incluye una capa lineal con 10 neuronas y una función de activación Softmax, adecuada para la clasificación en 10 categorías (dígitos del

0 al 9).

Además de la arquitectura de la red, configuramos un entorno de laboratorio para especificar los parámetros del experimento:

- Optimizador: Utilizamos el optimizador Adam.
- Función de pérdida: Empleamos la MAELoss (Mean Absolute Error Loss).
- Estrategia de entrenamiento: Clasificación.
- Conjunto de datos: Utilizamos el dataset MNIST.

El dataset se carga desde un archivo denominado "mnist.tsv" y se procesa en lotes de tamaño 10. La división de los datos se establece de la siguiente manera: 70 % para entrenamiento, 20 % para prueba y 10 % para validación. Finalmente, definimos el experimento identificado como `e626`.

```
Laboratory(epochs = 10, name = "MNIST EXAMPLE")
    Adam
    MAELoss
    ClassificationStrategy
    Dataset(name = "mnist.tsv", batchSize = 10)
        Split(train=0.7, test=0.2, validation=0.1)

    Experiment e626

Architecture LinearNeuralNetwork
    RecurrentSection
    Input(seqLength=28, x=28)
        Block
            LSTM(bidirectional=true, numLayers=4)
                EndSequence(reduce=flatten) > Output(x=512)
            Linear(bidirectional=true, numLayers=4) > Output(x=2000)
    LinearSection
        Block
            Linear > Output(x=500)
            ReLU
        Block
            Linear > Output(x=100)
            ReLU
        Block
            Linear > Output(x=10)
            Softmax
```

Ilustración 5.8: Red neuronal recurrente definida con el DSL de *Flogo*.

La manera de especificar redes recurrentes en *Flogo*, como se muestra en el ejemplo, proporciona una mayor legibilidad y claridad en comparación con las implementaciones tradicionales. Esta configuración permite definir la arquitectura de la red sin tener que lidiar con la complejidad inherente a la recurrencia clásica de las redes recurrentes.

5.4. Modelo para la estimación de la demanda

Este caso de uso fue desarrollado durante un internship de verano de 2023 en EIFER (*European Institute for Energy Research*), un centro de investigación colaborativo establecido para estudiar y desarrollar soluciones sostenibles en el ámbito de la energía. EIFER fue fundado como una colaboración entre el Instituto de Tecnología de Karlsruhe, (KIT) y Electricité de France (EDF), uno de los mayores productores y proveedores de electricidad en Europa. La misión de EIFER se centra en la transición hacia sistemas energéticos sostenibles, abarcando una variedad de áreas de investigación como la producción de energía renovable, métodos para mejorar el uso de la energía, la investigación en tecnologías de red inteligentes... Este centro lleva más de 20 años de colaboración entre la ciencia y la industria, contribuyendo significativamente a la transición energética europea.

Durante el *Internship* se trabajó con *microgrids*, sistemas energéticos locales que operan de manera autónoma. Estos *microgrids* son capaces de generar, distribuir y regular el flujo de electricidad de manera independiente, permitiéndoles funcionar sin conexión a la red principal. Los *microgrids* son ideales en comunidades aisladas donde el acceso a la red principal es limitado o inexistente, proporcionando una solución esencial para estas áreas remotas.

El *DSL Flogo* fue empleado para construir un modelo de predicción de la demanda eléctrica del *microgrid*, para estimar la cantidad de energía necesaria para un día específico en la villa. El modelo se alimentó con un conjunto de datos recolectados durante 578 días, incluyendo variables como la temperatura, la humedad, la hora del día, el mes. etc. Estas variables de entrada fueron analizadas para estimar la potencia requerida, expresada en kilovatios (kW).

La razón de usar redes neuronales en este proyecto radica en la capacidad de estas tecnologías para manejar y analizar interacciones complejas y no lineales entre múltiples variables. Las redes neuronales son especialmente adecuadas para identificar patrones y hacer predicciones precisas en situaciones donde las relaciones entre los datos no son aparentes o son demasiado complejas para modelos estadísticos tradicionales.

En la figura 5.9 se muestra la red neuronal diseñada, un perceptrón multicapa que transforma un conjunto de datos de entrada de dimensiones reducidas en una representación de dimensiones ampliadas, finalizando con un valor escalado que representa la cantidad de energía necesaria para ese conjunto de datos de entrada.

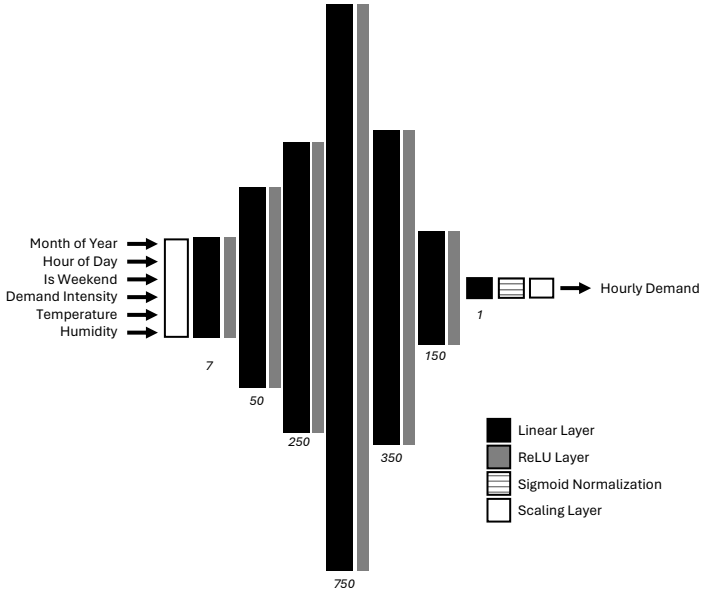

Ilustración 5.9: Perceptrón multicapa para la estimación de la demanda de un microgrid

Esta arquitectura demostró su efectividad para capturar y modelar las interacciones complejas entre los datos y la demanda eléctrica El modelo desarrollado logró un error absoluto medio (MAE) del 10.2 %. Demostrando el potencial de las redes neuronales para modelar eficazmente la demanda eléctrica en áreas con datos limitados.

Capítulo 6

Proceso de desarrollo

6.1. Metodología

En este proyecto, se han aplicado una serie de metodologías estructuradas para garantizar un desarrollo eficiente y efectivo.

6.1.1. Agile

Agile es una metodología de gestión de proyectos y desarrollo de software que enfatiza la flexibilidad, la colaboración y la entrega continua de valor. Se basa en un conjunto de principios y prácticas que permiten a los equipos adaptarse rápidamente a los cambios y mejorar continuamente el producto a lo largo del ciclo de vida del proyecto. Agile es especialmente útil en proyectos de carácter colaborativo y en entornos donde los requisitos pueden evolucionar con el tiempo.

En el desarrollo de *Flogo* se ha adoptado metodología *Agile* para gestionar el del proyecto de manera iterativa e incremental. Esto ha permitido que el equipo se adapte rápidamente a los cambios, además mejorar continuamente el producto a lo largo del ciclo de vida del proyecto. Algunas de las prácticas *Agile* que hemos implementado incluyen:

- **Scrum**: Se dividió el trabajo en *sprints* (unidad básica de trabajo) cortos y manejables, lo que permitió entregar incrementos funcionales del producto de manera regular.
- **Reuniones semanales**: Se mantuvieron reuniones con cierta frecuencia para revisar el progreso, identificar obstáculos y ajustar las prioridades según sea necesario.
- **Retrospectivas**: Realizamos retrospectivas al final de cada sprint para evaluar lo que funcionó bien y qué áreas necesitaban mejora, fomentando así un ciclo de retroalimentación constante.

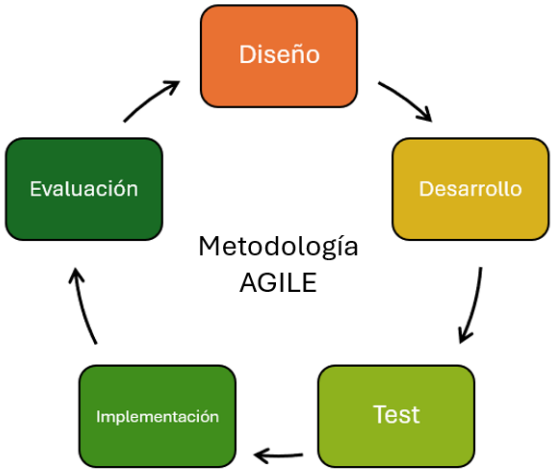

Ilustración 6.1: Fases de una metodología Agile

Este enfoque ha resultado realmente útil puesto que no conocíamos cuál sería la solución técnica que queríamos alcanzar desde el inicio del proyecto. A través de iteraciones de corta duración fuimos capaces de ir ideando y programando el software que cumpliese con todos los requisitos establecidos.

6.1.2. TDD

El desarrollo orientado a pruebas (*Test-Driven Development*, TDD) fue la metodología empleada para desarrollar el generador de código en nuestro proyecto. TDD es una técnica de desarrollo de software que enfatiza la escritura de pruebas automatizadas antes de escribir el código funcional. Este enfoque ayuda a asegurar la calidad y fiabilidad del software, garantizando que el código cumpla con los requisitos especificados desde el principio. El proceso TDD es el siguiente:

- **Escribir Pruebas Antes del Código**: Para cada nueva funcionalidad, primero escribimos una prueba que describía el comportamiento esperado.
- **Hacer que la prueba pase**: Luego implementamos el código necesario para pasar la prueba, asegurándonos de que cumpliera con los requisitos especificados.
- **Refactorización**: Una vez que las pruebas pasaban, refactorizamos el código para mejorar su estructura y legibilidad sin cambiar su comportamiento.

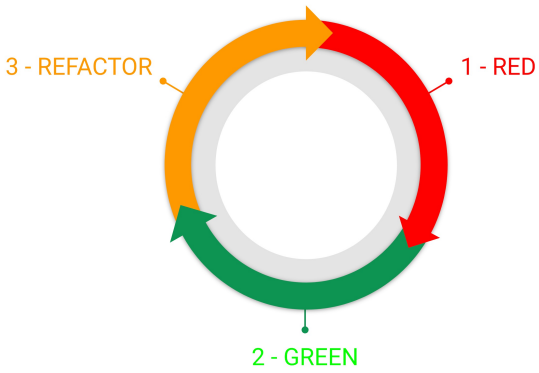

Ilustración 6.2: Fases de la metodología de TDD

Esta metodología es extremadamente útil cuando no se tiene claro cómo orientar el código o cuál debería ser el resultado final del código. En el caso del generador de código, escribir pruebas primero obligó a pensar en los requisitos y en el diseño de la funcionalidad antes de comenzar a implementar el código. Esto proporcionó una guía clara y específica para la implementación, asegurando que cada parte del generador de código cumpliera con las especificaciones y expectativas definidas. Dado que el generador de código es un módulo crítico, tener una dirección clara desde el principio ayudó a evitar ambigüedades y errores durante el desarrollo.

6.1.3. Gitflow

Gitflow es una metodología de branching diseñada para facilitar la gestión de versiones y el desarrollo colaborativo de software. Proporciona una estructura clara y organizada para manejar el desarrollo de características, corrección de errores y lanzamientos de versiones.

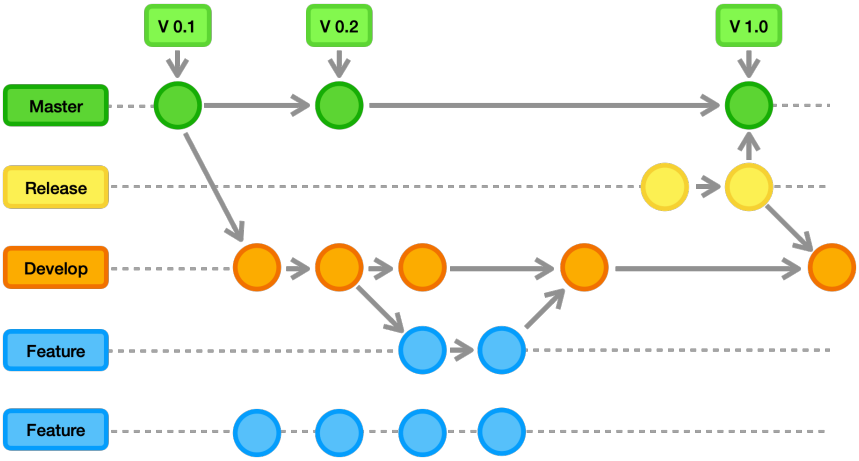

Ilustración 6.3: Fases de la metodología *Gitflow*

Gitflow se basa en el uso de varias ramas paralelas, cada una con un propósito específico,

lo que ayuda a mantener un flujo de trabajo ordenado y eficiente. *Gitflow* nos proporcionó una estructura clara para manejar el desarrollo de nuevas características y corrección de errores. Las principales ramas utilizadas en *Gitflow* fueron:

- **Main**: Contiene el código de producción estable.
- **Develop**: Utilizada para el desarrollo de nuevas características y funcionalidades.
- **Feature Branches**: Ramas creadas desde develop para trabajar en nuevas características. Una vez completadas y probadas, se fusionan de nuevo en develop.
- **Release Branches**: Ramas creadas para preparar una nueva versión de producción. Permiten realizar pruebas finales y ajustes menores antes de fusionarse en main.
- **Hotfix Branches**: Ramas creadas desde main para corregir errores críticos en producción. Una vez solucionados, se fusionan de nuevo en main y develop.

6.1.4. Semantic Versioning

El versionado semántico, comúnmente conocido como SemVer (*Semantic Versioning*), es un sistema de numeración para la asignación de versiones a software que busca comunicar de manera clara y concisa el impacto de los cambios realizados en el código.

El versionado semántico es una práctica esencial en el desarrollo de software que mejora la comunicación, la gestión de dependencias y la confianza en la estabilidad del software. Adoptar SemVer en los proyectos facilita la gestión de la configuración y la compatibilidad entre versiones, lo que es crucial para el desarrollo de software colaborativo y el mantenimiento a largo plazo.

SemVer utiliza un esquema de numeración de versiones en el formato MAJOR.MINOR.PATCH, donde cada uno de estos componentes tiene un significado específico:

- **MAJOR**: Este número se incrementa cuando se realizan cambios incompatibles con versiones anteriores. Por ejemplo, si un cambio en el software rompe la compatibilidad con la versión previa, se debe incrementar el número de la versión principal. Un ejemplo sería pasar de 1.0.0 a 2.0.0.
- **MINOR**: Este número se incrementa cuando se añaden funcionalidades nuevas de manera compatible con las versiones anteriores. Esto significa que los cambios no afectan el funcionamiento de la versión previa. Por ejemplo, si se añade una nueva función sin alterar las existentes, la versión podría pasar de 1.0.0 a 1.1.0.
- **PATCH**: Este número se incrementa cuando se realizan correcciones de errores o mejoras menores que no afectan la compatibilidad con las versiones anteriores. Por ejemplo, si se corrige un bug sin introducir nuevas características, la versión podría cambiar de 1.0.0 a 1.0.1.

6.2. Herramientas utilizadas

En el desarrollo de este proyecto, hemos recurrido a una variedad de herramientas que han sido fundamentales para garantizar la eficiencia, la calidad y la robustez de *Flogo*. La elección de estas herramientas se ha basado en su capacidad para manejar las complejidades inherentes al desarrollo de redes neuronales, facilitar la colaboración en equipo y mejorar la productividad general del proceso de desarrollo.

6.2.1. Java

Java es un lenguaje de programación compilado de propósito general, orientado a objetos, conocido por su portabilidad, robustez y extensibilidad. Ha sido ampliamente utilizado en el desarrollo de aplicaciones empresariales y científicas debido a sus características clave, como la gestión automática de memoria, el manejo de excepciones, y sus sólidas capacidades de concurrencia.

Una de las funcionalidades de Java más utilizadas en este proyecto es la reflexión (reflection). Reflection es una poderosa característica que permite al código inspeccionar y manipular las propiedades y comportamientos de los objetos en tiempo de ejecución.

6.2.2. Intellj IDEA

IntelliJ IDEA es un IDE (entorno de desarrollo integrado) que ofrece una serie de características avanzadas que han facilitado enormemente el proceso de desarrollo. Su potente sistema de autocompletado y sugerencias de código ha permitido escribir código más rápidamente y con menos errores. Además, la capacidad de refactorización automática ha sido crucial para mantener un código limpio y bien organizado, lo cual es esencial en proyectos complejos como este. La integración con sistemas de control de versiones, como Git, ha hecho que la colaboración entre los miembros del equipo sea más fluida y eficiente, permitiéndonos gestionar cambios y resolver conflictos de manera efectiva.

Otra ventaja significativa de IntelliJ IDEA es su robusto conjunto de herramientas de depuración y pruebas. La capacidad de establecer puntos de interrupción (breakpoints), entre otras, has sido crucial para identificar y resolver problemas rápidamente. Además, la integración con marcos de pruebas como JUnit ha facilitado la implementación y ejecución de pruebas automatizadas, asegurando que la compilación del código de *Flogo* funcione correctamente y cumplan con los requisitos especificados.

6.2.3. Intino

Intino es un método de Ingeniería Dirigida por Modelos (MDE) que emplea un enfoque único para el modelado, centrado en simplificar el proceso de desarrollo de Lenguajes de

Modelado Específicos de Dominio (DSMLs). Este método no solo reduce el esfuerzo necesario para desarrollar DSMLs, sino que también elimina la necesidad de tener una experiencia especializada en desarrollo de lenguajes.

Tres características fundamentales distinguen a Intino de los enfoques MDE actuales. En primer lugar, propone la implementación de un modelo único y unificado que integra los tres niveles de abstracción, en contraste con la práctica tradicional de utilizar tres modelos distintos. Esta aproximación garantiza una representación conectada del dominio a través de diferentes niveles de abstracción. En segundo lugar, en lugar de depender de un solo DSML, Intino propone tener un DSML distinto para cada nivel de abstracción. Finalmente, Intino introduce un proceso de modelado por etapas, en el cual, después de modelar en niveles más abstractos, se genera automáticamente un DSML para modelar el siguiente nivel de abstracción.

El dominio y sus niveles de abstracción son fundamentales en este enfoque. Un dominio se refiere a un área específica de interés dentro de la realidad más amplia, y la determinación de un dominio se ve influenciada por los objetivos, perspectivas o preguntas específicas que se buscan responder. A través de un marco analítico, se puede visualizar la realidad para determinar el dominio, lo que permite una exploración enfocada. En la práctica, esto significa identificar no solo las entidades dentro de un dominio, sino también los conceptos que las organizan y los metaconceptos que estructuran el dominio en un nivel más alto de abstracción.

6.2.4. Tara

Tara es el lenguaje madre diseñado para soportar los constructos analíticos propuestos en la investigación asociada a Intino. Tara cumple con los requisitos de un lenguaje madre al ofrecer una sintaxis universal y un léxico y reglas semánticas fundamentales que pueden personalizarse para dominios específicos. El mogram es la estructura sintáctica central en la gramática de Tara, diseñada para el modelado de entidades, conceptos y metaconceptos. Cada mogram se estructura en dos secciones principales: el encabezado obligatorio y el cuerpo opcional. El encabezado incluye la firma del mogram, que contiene el tipo de mogram y puede incluir elementos opcionales como cardinalidad, constructor y nombre. También pueden incluirse superposiciones de especialización, designaciones de facetas u objetivos de facetas, y anotaciones para introducir restricciones semánticas. El cuerpo del mogram comienza en una nueva línea e indentado para indicar su alcance, y se utiliza para declarar propiedades y establecer relaciones con otros mograms, permitiendo la declaración de propiedades descriptivas y prescriptivas, así como generalización, extensión de facetas o composición.

En términos de constructos analíticos, un mogram $\mathcal{M}^1$ representa una entidad, expresando la concretización de un concepto; un mogram $\mathcal{M}^2$ representa un concepto, expresando tanto la abstracción de un concepto como la concretización de un metaconcepto; y un mogram $\mathcal{M}^3$ representa un metaconcepto, expresando la abstracción de metaconceptos a partir de conceptos. Otros constructos relacionales se expresan mediante especialización (indicada en el encabezado con 'extends'), generalización (indicada en el cuerpo con 'sub'), composición (expresada directamente en el cuerpo del mogram o con la palabra clave 'has'), designación de

facetas (indicada en el encabezado con 'as'), objetivo de faceta (indicada con 'on'), y extensión de faceta (indicada en el cuerpo con 'facet'). Las restricciones semánticas se manejan a través de estructuras sintácticas que permiten expresar cardinalidad, coherencia y unidad de medida para las propiedades.

6.2.5. ItRules

ItRules es un sistema de generación de código basado en plantillas que emplea un enfoque de programación lógica, inspirado en los principios de separación de lógica y control. En lugar de utilizar sentencias de control tradicionales dentro de las plantillas, *ItRules* define un conjunto de reglas que especifican las condiciones bajo las cuales se generan ciertas salidas. El motor de plantillas de *ItRules* se encarga de gestionar el control de forma externa, evaluando las reglas y aplicando las transformaciones necesarias durante la ejecución. Esto permite una especificación clara y declarativa de las plantillas, eliminando la complejidad asociada con las sentencias de control dentro de las mismas.

ItRules incluye un lenguaje específico del dominio (*DSL*) diseñado para definir plantillas de manera concisa y expresiva. Para definir una plantilla con este *DSL*, se crean reglas que consisten en una condición y un patrón de salida. Cada regla comienza con la palabra clave 'def' seguida de una condición lógica que determina cuándo se aplica la regla. El patrón de salida, contiene el texto a generar e incluye placeholders para insertar datos dinámicos. Los placeholders, marcados con un símbolo de dólar '$', se utilizan para señalar puntos de inserción de datos y pueden incluir formatters que transforman los datos antes de su inserción.

Este forma de generar código ofrece varias ventajas significativas: proporciona claridad y legibilidad al eliminar la necesidad de sentencias de control complejas dentro de las plantillas, lo que resulta en plantillas más fáciles de leer y mantener; mejora la modularidad permitiendo definir reglas independientes y reutilizables; ofrece flexibilidad a través de placeholders y formatters que permiten personalizar y adaptar las plantillas a diversos escenarios; y es extensible, ya que permite agregar nuevos formatters y predicados de condición para satisfacer necesidades específicas. Además, *ItRules* mejora la productividad al simplificar la creación y mantenimiento de plantillas, reduciendo así el tiempo dedicado a la generación manual de código y disminuyendo la probabilidad de errores, especialmente en proyectos complejos.

6.2.6. JUnit

JUnit es una de las herramientas de pruebas unitarias más populares y ampliamente utilizadas en el ecosistema Java. Su principal propósito es facilitar la escritura y ejecución de pruebas unitarias, permitiendo a los desarrolladores verificar que cada unidad individual de código funcione correctamente. JUnit proporciona un marco estructurado para definir y organizar pruebas, lo que ayuda a mantener un código de prueba limpio y coherente. Además, JUnit incluye diversas anotaciones que simplifican la configuración y limpieza del entorno de pruebas, mejorando la eficiencia del proceso de pruebas.

Una de las mayores ventajas de JUnit es su integración con entornos de desarrollo integra-

dos (IDEs) y sistemas de integración continua (CI). Esto permite ejecutar automáticamente las pruebas como parte del proceso de compilación, asegurando que cualquier cambio en el código no introduzca errores. JUnit también ofrece soporte para pruebas parametrizadas, permitiendo ejecutar el mismo conjunto de pruebas con diferentes datos de entrada, lo que aumenta la cobertura de pruebas y ayuda a identificar casos borde y errores potenciales. En nuestro proyecto, JUnit ha sido una herramienta esencial para garantizar la estabilidad y funcionalidad del sistema, permitiéndonos detectar y corregir errores de manera temprana y eficiente.

6.2.7. AssertJ

AssertJ es una biblioteca de aserciones para Java que proporciona una forma fluida y legible de escribir pruebas. A diferencia de las aserciones tradicionales de JUnit, AssertJ permite a los desarrolladores encadenar métodos de manera intuitiva, lo que mejora la claridad y expresividad de las pruebas. Esto hace que el código de prueba sea más fácil de leer y entender, lo que a su vez facilita el mantenimiento y la ampliación de las pruebas a lo largo del tiempo. Por ejemplo, en lugar de escribir múltiples aserciones dispersas, AssertJ permite combinarlas en una sola línea, proporcionando mensajes de error más detallados y específicos.

Una de las principales ventajas de AssertJ es su extensa API, que ofrece una amplia variedad de métodos de aserción para diferentes tipos de datos, incluyendo colecciones, mapas, y excepciones. Además, AssertJ es altamente extensible, lo que permite a los desarrolladores crear sus propias aserciones personalizadas. Esta flexibilidad es particularmente útil en proyectos grandes y complejos, donde las pruebas necesitan adaptarse a requisitos específicos. En nuestro proyecto, AssertJ ha sido fundamental para asegurar la calidad del código, permitiéndonos escribir pruebas claras, concisas y efectivas que garantizan que cada componente funcione según lo esperado.

6.2.8. Python

Python ha sido una de las herramientas clave debido a su amplio uso en el desarrollo de redes neuronales. Python es un lenguaje de programación interpretado, de alto nivel, conocido por su sintaxis clara y legible, lo cual facilita la rápida implementación de prototipos y el desarrollo de software complejo. La simplicidad del lenguaje permite a los desarrolladores centrarse en resolver problemas, en lugar de lidiar con la complejidad sintáctica, lo que resulta en una mayor eficiencia y productividad.

El lenguaje soporta múltiples paradigmas de programación, incluyendo la programación orientada a objetos o la programación funcional. Esta flexibilidad permite a los desarrolladores elegir el estilo de programación que mejor se adapte a sus necesidades específicas. Además, Python es altamente extensible, permitiendo la integración con otros lenguajes y herramientas, lo que facilita la incorporación de código y bibliotecas externas para ampliar su funcionalidad.

6.2.9. Pycharm

PyCharm es un IDE que ha sido fundamental para mejorar nuestra productividad y eficiencia en los frameworks de *Flogo*. PyCharm ofrece una serie de características avanzadas que han facilitado enormemente el proceso de desarrollo. Su potente sistema de autocompletado y sugerencias de código nos ha permitido escribir código más rápidamente y con menos errores. Además, la capacidad de refactorización automática ha sido crucial para mantener un código limpio y bien organizado, lo cual es esencial en proyectos complejos como este. La integración con sistemas de control de versiones, como Git, ha hecho que la colaboración entre los miembros del equipo sea más fluida y eficiente, permitendo gestionar cambios y resolver conflictos de manera efectiva.

6.2.10. PyTorch

En el desarrollo de este proyecto, PyTorch ha sido una de las herramientas más importantes debido a su flexibilidad y potencia en el ámbito del aprendizaje profundo. PyTorch es un marco de código abierto para el aprendizaje automático basado en bibliotecas que proporciona una interfaz optimizada para desarrollar y entrenar redes neuronales.

Una de las principales razones que nos llevaron a elegir PyTorch sobre otros frameworks fue la gran capacidad de personalización que ofrece en el proceso de creación y entrenamiento de modelos. A diferencia de otras opciones disponibles, PyTorch permite una programación a un nivel mucho más bajo, lo que proporciona una mayor flexibilidad para adaptarnos a nuestras preferencias y necesidades específicas. Esta flexibilidad nos ha permitido ajustar y optimizar los modelos de manera más precisa, aprovechando al máximo las capacidades del framework para nuestros objetivos de desarrollo.

6.2.11. Github

GitHub ha sido una herramienta esencial para la gestión del código fuente y la colaboración entre los miembros del equipo. GitHub es una plataforma basada en Git que ofrece control de versiones y una amplia gama de funcionalidades colaborativas que han sido fundamentales para el éxito de nuestro proyecto.

Una de las mayores ventajas de GitHub es su capacidad para facilitar la colaboración en equipo. Las funcionalidades de pull requests y revisiones de código (code reviews) han permitido que los miembros del equipo revisen y discutan cambios antes de integrarlos en las diferentes ramas. Este proceso no solo ha mejorado la calidad del código, sino que también ha fomentado un ambiente de trabajo más colaborativo. Las discusiones y comentarios en las diferentes secciones del código han sido vitales para resolver problemas y mejorar las implementaciones.

6.2.12. HTML

HTML (*HyperText Markup Language*) es el lenguaje estándar utilizado para crear y diseñar páginas web. A diferencia de LaTeX, HTML se basa en una estructura de etiquetas y atributos que permiten definir el contenido y la estructura de un documento web. Es fundamental en el desarrollo web, ya que permite la integración de texto, imágenes, enlaces, formularios y otros elementos interactivos en una página.

HTML es fácil de aprender y utilizar, y es compatible con la mayoría de los navegadores web. Además, su funcionalidad se puede ampliar mediante el uso de hojas de estilo en cascada (CSS) para el diseño y JavaScript para la interactividad. La combinación de HTML con CSS y JavaScript permite crear sitios web dinámicos y visualmente atractivos. En el proyecto, se ha utilizado HTML para la creación de la documentación.

6.2.13. Overleaf

Overleaf es una plataforma en línea para la edición colaborativa de documentos LaTeX. Proporciona un entorno de edición intuitivo y accesible que permite a los usuarios trabajar en documentos LaTeX desde cualquier lugar con acceso a internet. Overleaf combina las potentes capacidades de LaTeX con herramientas colaborativas avanzadas, lo que la convierte en una opción ideal para equipos de desarrollo y proyectos académicos.

La interfaz de usuario es intuitiva y amigable, con herramientas de ayuda y plantillas que simplifican la creación de documentos complejos. Overleaf también ofrece documentos realizados en colaboración, lo que permite a los miembros del equipo unirse al proyecto fácilmente y contribuir sin necesidad de instalar software adicional. Esta accesibilidad ha permitido a todos los miembros del equipo participar plenamente en la redacción y revisión de los diferentes documentos, mejorando la calidad y cohesión del trabajo final.

6.3. Cronología

En este apartado, se describe detalladamente el procedimiento seguido durante los meses de trabajo en este proyecto, destacando las herramientas y metodologías aplicadas para alcanzar los resultados finales. Esta cronología muestra los pasos seguidos y cómo se ha implementado cada uno de los conceptos explicados en este capítulo.

La motivación de este TFG radica con una problemática recurrente observada a lo largo de los cuatro años de estudio del grado: el proceso de creación de modelos basados en modelos de redes neuronales es tedioso de estudiar, realizar y mantener. Con esta premisa, los primeros sprints se dedicaron a validar esta hipótesis respecto a la limitada aplicación de principios de ingeniería de software en el ámbito de la inteligencia artificial.

6.3.1. Iniciación y planificación

Esta fase fue crucial para establecer las bases del proyecto. Se comenzó estableciendo objetivos claros para el *DSL* y cada *framework*, y se definió el alcance del proyecto y sus limitaciones. Además, se elaboró un cronograma detallado con hitos y fechas límite, y se asignaron recursos y definieron roles y responsabilidades para asegurar una gestión organizada del proyecto.

Inicialmente, comenzamos a desarrollar de manera conjunta todos los componentes del proyecto. Sin embargo, debido a la complejidad que surgió durante el proceso, decidimos que lo mejor sería que cada componente tuviera un especialista dedicado. Así, se asignaron roles específicos: un especialista se centró en el *DSL* y el generador de código, otro en el *framework* estructural y otro en el *framework* operacional. Esta división de responsabilidades permitió un enfoque más profundo y especializado en cada área, asegurando que cada componente se desarrollara de manera eficiente y efectiva.

Una vez definidos los roles de cada miembro del equipo, se decidió que los *frameworks* estructural y operacional se desarrollarían en *Python* utilizando *PyTorch*, debido a su flexibilidad y potencia en el ámbito del aprendizaje profundo. Por otro lado, se determinó que tanto el generador de código como el *DSL* se implementarían en Java, debido a la existencia de tecnologías como ItRules, JUnit, Tara o Intino. Esta combinación de tecnologías fue clave para optimizar el desarrollo y garantizar la integración fluida de todos los componentes del sistema *Flogo*.

6.3.2. Diseño y desarrollo

Durante la fase de diseño y desarrollo, realizamos reuniones periódicas para revisar el progreso de cada componente, identificar puntos débiles en el código y discutir posibles mejoras. Estas reuniones también se centraron en aspectos de diseño para asegurarnos de que todos estuviéramos satisfechos con los componentes desarrollados y que se mantuviera una coherencia en el diseño general del sistema.

En esta fase, se creó el diseño arquitectónico del *DSL*. Se implementó un enfoque iterativo y ágil de desarrollo, utilizando *Gitflow* para la gestión de ramas. Cada componente se desarrolló en ciclos, con revisiones y ajustes regulares. La integración y pruebas continuas jugaron un papel fundamental, utilizando herramientas como JUnit o AssertJ para asegurar la calidad del software.

La colaboración constante y las revisiones detalladas de cada componente permitieron mejorar el código, resolver problemas de diseño y asegurar que todos los miembros del equipo estuvieran alineados con los objetivos del proyecto. Esta estructura iterativa no solo facilitó la incorporación de mejoras continuas, sino que también aseguró que cada iteración aportara valor al desarrollo del sistema.

6.3.3. Revisión y ajustes

En esta fase se procedió con la integración de los diferentes componentes del sistema para asegurar que todos funcionaran como se había previsto. Nos centramos en unir el DSL y el generador de código con los *frameworks* estructural y operacional, verificando la correcta comunicación y la interoperabilidad. Las pruebas de integración fueron esenciales para identificar y resolver conflictos que surgieron al combinar estos componentes.

También discutimos los desafíos encontrados en la integración y propusimos soluciones para superar estos obstáculos. Este proceso iterativo de prueba y ajuste no solo mejoró la funcionalidad individual de cada componente, sino que también garantizó que el sistema en su conjunto operara de manera coherente y eficiente.

La colaboración entre los responsables de cada componente fue vital para alinear los objetivos y asegurar que todas las partes del sistema se integraran sin problemas. Esta fase permitió asegurar que el sistema final no solo cumpliera con los requisitos especificados, sino que también ofreciera un rendimiento robusto y confiable, alineado con las expectativas del proyecto.

6.3.4. Documentación y capacitación

La fase final del proyecto se enfocó en la elaboración de documentación completa para el DSL. Se aseguró que la documentación fuera clara, precisa y fácil de seguir para los usuarios finales. Para lograr esto, utilizamos LaTeX y *Markdown* para generar documentación tanto para el repositorio de GitHub como para las memorias finales del proyecto. LaTeX se utilizó principalmente para las memorias finales y *Markdown* se empleó para la documentación del repositorio de GitHub, facilitando una presentación limpia y estructurada.

En resumen, el proyecto se desarrolló de manera iterativa y colaborativa, con un enfoque en la calidad y la eficiencia. La documentación detallada aseguró que el sistema fuera robusto, fácil de usar y sostenible a largo plazo.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Resultados

En este trabajo hemos presentado *Flogo*, un lenguaje específico de dominio (*DSL*) que simplifica la creación y manejo de redes neuronales desde la concepción hasta tener el modelo final entrenado. El objetivo principal de automatizar el proceso de diseño y entrenamiento de redes neuronales mediante la abstracción de complejidades inherentes a *frameworks* como *PyTorch* ha sido alcanzado. En esta primera versión de *Flogo* hemos dado soporte a varios tipos de redes neuronales:

- **Perceptrones multi-capa**: Usadas en tareas de regresión y clasificación.
- **Redes Neuronales Convolucionales** (Convolutional Neural Networks - *CNN*): Usadas en reconocimiento de imágenes y video, y análisis de series temporales.
- **Redes Neuronales Recurrentes** (*Recurrent Neural Networks - RNN*): Usadas para procesar secuencias de datos como el lenguaje natural o series temporales.
- **Autoencoders**: Usadas para la reducción de dimensionalidad y el aprendizaje de representaciones eficientes de los datos de entrada sin supervisión.

Una de las características más interesante que hemos implementado en *Flogo* es la capacidad de parametrizar las arquitecturas de las redes neuronales. Tradicionalmente, el desarrollo de redes neuronales implica un proceso de prueba y error para determinar la configuración óptima de la arquitectura. Esta búsqueda puede ser intensiva en tiempo y recursos, ya que requiere múltiples iteraciones de entrenamiento con diferentes configuraciones. La capacidad de parametrizar arquitecturas en *Flogo* aborda este desafío permitiendo que el modelo se ajuste y optimice automáticamente durante el entrenamiento.

Al introducir parámetros ajustables en la definición de la arquitectura de la red, *Flogo* hace que el entrenamiento sea más automatizado. Este enfoque no solo acelera el proceso de encontrar una arquitectura adecuada, sino que también mejora la capacidad del modelo para adaptarse a los datos sin intervención manual. En lugar de requerir que un desarrollador ajuste manualmente los parámetros de la red, *Flogo* puede probar automáticamente diferentes

configuraciones y ver cual es la estructura mejor.

Por ejemplo, los usuarios pueden especificar que se prueben diferentes funciones de activación, como la función sigmoide o ReLU, durante el proceso de entrenamiento. *Flogo* evalúa automáticamente cuál de estas funciones produce los mejores resultados en términos de rendimiento del modelo, ajustando la arquitectura de acuerdo a estos hallazgos. Esto elimina la necesidad de manualmente probar y seleccionar funciones de activación, optimizando el flujo de trabajo de desarrollo y permitiendo que el modelo se adapte mejor a las peculiaridades de los datos. Otro ejemplo, es la posibilidad de parametrizar el número de entradas en las capas. Esto significa que el modelo puede ajustar su propia estructura interna. Al permitir que el modelo experimente con diferentes tamaños de capas, se puede explorar el espacio de soluciones potenciales, aumentando la posibilidad de encontrar una configuración que maximice el rendimiento.

Estas características de parametrización no solo aumentan la automatización en el diseño de modelos, sino que también potencian al programador cuando tiene que hacer sutiles cambios en la arquitectura de los modelos. Esto es especialmente útil en situaciones donde la arquitectura no está completamente definida y puede evolucionar con el tiempo, o donde distintas tareas requieren ajustes menores en la arquitectura de un mismo modelo.

7.2. Aportaciones

Los resultados obtenidos demuestran que *Flogo* no solo reduce el tiempo de desarrollo y los errores, sino que también proporciona una plataforma flexible y robusta capaz de adaptarse a los cambios tecnológicos rápidos y a las exigencias del mercado. Los desafíos técnicos enfrentados durante el desarrollo nos han permitido mejorar la versatilidad del DSL, preparándolo para futuras ampliaciones que podrían incluir soporte para una gama más amplia de lenguajes y *frameworks*. En este sentido, los principales resultados han sido:

- **Abstracción de la complejidad**. *Flogo* abstrae efectivamente la complejidad técnica de *frameworks* como *PyTorch* y *TensorFlow*, permitiendo a usuarios de todos los niveles técnicos diseñar y modelar eficazmente redes neuronales. Esto democratiza el uso de tecnologías avanzadas, haciendo el campo del aprendizaje automático más accesible.
- **Automatización del proceso de desarrollo**. El *DSL* permite una definición clara y sencilla de la arquitectura de una red neuronal, su entrenamiento y validación, automatizando muchos de los procesos que tradicionalmente requerían intervención manual detallada y propensa a errores.
- **Optimización del tiempo de desarrollo y reducción de errores**. *Flogo* reduce significativamente el tiempo de desarrollo de modelos de aprendizaje automático y minimiza los errores comunes en la codificación y configuración de estos modelos. Esto no solo mejora la eficiencia sino que también incrementa la fiabilidad de los proyectos de aprendizaje automático.
- **Facilidad de adaptación y evolución tecnológica**. La arquitectura modular y escalable de *Flogo* permite que el *DSL* se adapte fácilmente a cambios en las tecnologías

subyacentes y a las necesidades emergentes del mercado, asegurando que la herramienta permanezca relevante y eficaz frente a los rápidos avances tecnológicos.

- **Exploración rápida de arquitecturas**. *Flogo* permite a los usuarios explorar diferentes arquitecturas de redes neuronales de manera más rápida mediante laboratorios y experimentos, lo que facilita la innovación y el descubrimiento de nuevas soluciones.
- **Impacto en la colaboración interdisciplinaria**. *Flogo* fomenta una colaboración más estrecha entre los desarrolladores de software, expertos en dominios específicos y académicos, promoviendo la innovación a través de sus herramientas intuitivas y su diseño accesible.
- **Estandarización de la comunicación**. Un *DSL* con constructos específicos permite una mejor comunicación entre personas, ya que proporciona un lenguaje común y claro. Esto asegura que todos los miembros del equipo, independientemente de su nivel técnico, puedan colaborar de manera más efectiva en el desarrollo y el entrenamiento de redes neuronales.

7.3. Aprendizaje personal

Desde una perspectiva personal, la participación en este proyecto ha sido una experiencia enriquecedora y formativa. Este proyecto ha ofrecido la oportunidad única de trabajar en un entorno colaborativo que imita de cerca la dinámica de proyectos reales en el ámbito profesional. La colaboración entre compañeros no solo ha mejorado nuestras habilidades de comunicación y trabajo en equipo, sino que también ha proporcionado una valiosa comprensión de cómo diferentes perspectivas y habilidades pueden unirse para resolver problemas complejos y alcanzar un objetivo común. La sinergia y la comunicación efectiva entre los miembros del equipo son claves para el éxito de la integración que tuvimos que realizar, demostrando que el trabajo colaborativo puede potenciar significativamente los resultados de un proyecto.

La interacción continua y el intercambio de ideas con mis compañeros durante el proyecto han sido fundamentales para mi desarrollo personal y profesional. Cada uno de nosotros aportó habilidades y conocimientos diferentes que, combinados, nos permitieron superar desafíos técnicos y lograr una integración exitosa. Este trabajo en equipo no solo mejoró la calidad del proyecto, sino que también nos preparó mejor para futuros proyectos colaborativos en el ámbito profesional.

Además, trabajar en el diseño de un *DSL* ha sido particularmente interesante. Esta experiencia ha ampliado mi comprensión de cómo los lenguajes de programación se desarrollan, proporcionando herramientas más eficientes y accesibles para tareas no convencionales. El enfrentar desafíos técnicos, junto con la necesidad de adaptarse y aprender sobre la marcha, ha imitado la realidad del desarrollo de software y la innovación tecnológica en la industria. Estas experiencias no solo han fortalecido mis habilidades técnicas, sino que también han mejorado mi capacidad de innovación.

7.4. Trabajo futuro

Nuestros objetivos futuros están orientados hacia la expansión y la innovación en varias áreas clave de las redes neuronales y el aprendizaje automático:

- **Desarrollo de nuevos tipos de redes neuronales**. Nos proponemos explorar y desarrollar nuevos tipos de arquitecturas de redes neuronales que puedan abordar problemas más complejos y específicos. Esto incluye tanto la investigación de nuevas topologías de red como la adaptación de arquitecturas existentes para mejorar su eficacia y eficiencia en diferentes tipos de tareas de aprendizaje automático.
- **Incorporación de métodos avanzados de aprendizaje**. Planeamos implementar técnicas avanzadas como el *Fine Tuning* y el *Transfer Learning*. El *Fine Tuning* permitirá ajustar modelos preentrenados a nuevos problemas con un menor requerimiento de datos, lo cual es especialmente útil en dominios donde los conjuntos de datos son pequeños o difíciles de recopilar. El *Transfer Learning* facilitará la reutilización de conocimientos adquiridos en una tarea para mejorar el aprendizaje en otra tarea relacionada, haciendo así que el proceso de entrenamiento sea más eficiente.
- **Avance en arquitecturas parametrizadas**. Continuaremos desarrollando y refinando el concepto de arquitecturas parametrizadas. Esto implicará crear modelos que puedan modificar su estructura durante el entrenamiento basados en algoritmos genéticos, permitiendo una mayor flexibilidad y adaptabilidad del modelo a las características específicas del problema.
- **Optimización**. Se pueden introducir mecanismos automatizados de optimización que operen dentro del espacio de parámetros del modelo. Estos mecanismos buscarán activamente los parámetros óptimos durante el proceso de entrenamiento, aplicando técnicas de optimización como algoritmos genéticos o métodos de gradiente. El objetivo es automatizar la búsqueda y ajuste de hiperparámetros para maximizar el rendimiento del modelo sin intervención manual extensiva.
- **Cursos de Formación física y online**. Además de los desarrollos técnicos, también queremos impulsar la capacitación en el uso de estas nuevas herramientas y técnicas. Planeamos ofrecer cursos de formación tanto físicos como online para compartir conocimientos y habilidades. Estos cursos estarán diseñados para diferentes niveles de experiencia, desde principiantes hasta expertos, con el objetivo de crear una comunidad.

Apéndices

Apéndice A

Propiedades de los constructos

A.1. Optimizers de *Flogo*

En todos los optimizadores de *Flogo* se debe establecer el learning rate. Éstos son:

- **SGD** [34]: Este es el método clásico de optimización en aprendizaje profundo, que actualiza los parámetros del modelo de manera iterativa ajustándose a los datos de forma estocástica, es decir, tomando muestras aleatorias de datos en cada actualización. Se puede especificar el momentum, el momentum decay y el weight decay.

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta_t)$$

- **SGD with Nesterov Momentum** [34]: Esta variante de SGD incorpora el momentum de Nesterov, una técnica que anticipa los pasos futuros del gradiente para hacer ajustes más informados durante el entrenamiento. Esto permite acelerar la convergencia hacia el mínimo del espacio de búsqueda y así mejorar la eficiencia del algoritmo, especialmente en superficies de error complejas o escarpadas. Se puede especificar el peso del momentum, el momentum decay y el weight decay.

$$v_{t+1} = \mu v_t + \eta \nabla_\theta J(\theta_t - \mu v_t)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

- **Adam** [34]: Este optimizador ajusta las tasas de aprendizaje de cada parámetro individualmente mediante el uso de estimaciones de primer y segundo momentos de los gradientes. Esto permite que Adam maneje diferentes escalas de parámetros de manera más efectiva y se adapte mejor a las características del problema, lo que generalmente resulta en una convergencia más rápida y estable en comparación con otros métodos de optimización. Se puede especificar las betas, el weight decay y un epsilon para evitar divisiones por cero.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
$$

- **ASGD** [38]: Es una variante del tradicional SGD, donde se toma un promedio de los parámetros a lo largo del tiempo. ASGD suaviza las fluctuaciones en la actualización de los parámetros debido al uso de promedios, lo que ayuda a reducir la varianza y potencialmente mejora la estabilidad y el rendimiento del modelo al aproximarse al final de la optimización, especialmente en escenarios donde los datos de entrenamiento son ruidosos o altamente variables. Se puede especificar el point to start average, la media, el alpha, el learning rate decay y el weight decay.

$$
\begin{aligned}
\theta_{t+1} &= \theta_t - \eta \nabla_\theta J(\theta_t) \\
\bar{\theta}_{t+1} &= \frac{t\bar{\theta}_t + \theta_{t+1}}{t+1}
\end{aligned}
$$

- **RMSProp** [34]: Este optimizador modifica y adapta las tasas de aprendizaje de cada parámetro utilizando un promedio móvil del cuadrado de los gradientes. Ayuda a evitar los problemas de las tasas de aprendizaje que no se adaptan bien a todas las partes del espacio de parámetros. Este enfoque es especialmente eficaz para abordar el problema del descenso de gradientes en minicapas muy empinadas o en valles planos. Se puede especificar el momentum, el alpha, el epsilon y el weight decay.

$$
\begin{aligned}
v_t &= \beta v_{t-1} + (1-\beta)(\nabla_\theta J(\theta_t))^2 \\
\theta_{t+1} &= \theta_t - \eta \frac{\nabla_\theta J(\theta_t)}{\sqrt{v_t} + \epsilon}
\end{aligned}
$$

- **Adagrad** [34]: Este optimizador ajusta de manera adaptativa la tasa de aprendizaje de cada parámetro basándose en la frecuencia de sus actualizaciones durante el entrenamiento. Incrementa la eficiencia de los parámetros menos frecuentes y disminuye la de los más frecuentes, lo que lo hace particularmente adecuado para tratar con datos dispersos y características que aparecen raramente, asegurando que todos los parámetros sean actualizados de manera justa y efectiva. Se puede especificar el learning rate decay, el epsilon, el weight decay y el initial accumulator.

$$
\begin{aligned}
G_t &= G_{t-1} + (\nabla_\theta J(\theta_t))^2 \\
\theta_{t+1} &= \theta_t - \eta \frac{\nabla_\theta J(\theta_t)}{\sqrt{G_t} + \epsilon}
\end{aligned}
$$

- **Adadelta** [34]: Es una extensión del método Adagrad que busca resolver su problema de disminución rápida de la tasa de aprendizaje. Adadelta hace esto al restringir la acumulación de los gradientes cuadrados a una ventana de tamaño fijo, usando un promedio móvil exponencial sin la necesidad de seleccionar una tasa de aprendizaje global. Esto permite que Adadelta continúe aprendiendo incluso cuando se han realizado muchas actualizaciones. Se puede especificar el rho, el epsilon y el weight decay.

$$
\begin{aligned}
E[g^2]_t &= \rho E[g^2]_{t-1} + (1-\rho)(\nabla_\theta J(\theta_t))^2 \\
\Delta\theta_t &= -\frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} \nabla_\theta J(\theta_t) \\
E[\Delta\theta^2]_t &= \rho E[\Delta\theta^2]_{t-1} + (1-\rho)(\Delta\theta_t)^2 \\
\theta_{t+1} &= \theta_t + \Delta\theta_t
\end{aligned}
$$

- **AdamW** [24]: Esta variante del optimizador Adam modifica la forma en que se aplica la regularización de los pesos. Permite aplicar la regularización de manera más efectiva y directa durante el proceso de optimización, lo que puede conducir a una mejor convergencia en algunos casos, especialmente en redes profundas y complejas. Se puede especificar las betas, el weight decay y si se pretende usar con amsgrad.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \eta\lambda\theta_t
\end{aligned}
$$

- **AMSGrad** [33]: Es una variante del optimizador Adam diseñada para actualizar las tasas de aprendizaje adaptativas de cada parámetro, asegurando que el denominador en la actualización de los parámetros nunca decrezca a lo largo del tiempo. Se pueden especificar las betas, el weight decay y el epsilon.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2 \\
\hat{v}_t &= \text{máx}(\hat{v}_{t-1}, v_t) \\
\theta_{t+1} &= \theta_t - \eta \frac{m_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
$$

- **SparseAdam** [18]: Es una variante del popular optimizador Adam diseñada específicamente para manejar gradientes dispersos, que son comunes en aplicaciones como el procesamiento del lenguaje natural (NLP) y la recomendación de sistemas, donde solo una pequeña fracción de parámetros se actualiza en cada iteración.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
$$

- **CenteredRMSProp** [39]: Es una variante del optimizador RMSProp que incorpora la media de los gradientes en el cálculo de la raíz cuadrada media (RMS). Esta inclusión ayuda a estabilizar la escala de los gradientes, proporcionando un enfoque más centrado y robusto. A diferencia de RMSProp, que solo considera la magnitud de los gradientes, CenteredRMSProp también tiene en cuenta la media, lo que puede reducir la oscilación de los gradientes y mejorar la convergencia.

$$
\begin{aligned}
g_t &= \nabla_\theta J(\theta_t) \\
g_{\text{mean}_t} &= \beta_1 g_{\text{mean}_{t-1}} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\theta_{t+1} &= \theta_t - \eta \frac{g_t}{\sqrt{v_t - g_{\text{mean}_t}^2 + \epsilon}}
\end{aligned}
$$

- **Adamax** [34]: Es una variante de Adam que emplea la norma infinita (norma máxima) en lugar de la norma L2. Esta modificación permite que Adamax sea más robusto frente a grandes variaciones en los gradientes.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \\
u_t &= \text{máx}(\beta_2 u_{t-1}, |\nabla_\theta J(\theta_t)|) \\
\theta_{t+1} &= \theta_t - \eta \frac{m_t}{u_t + \epsilon}
\end{aligned}
$$

- **NAdam** [34]: Combina los beneficios de Adam y Nesterov Accelerated Gradient (NAG), proporcionando un enfoque más refinado y eficiente para la optimización. NAdam ajusta los parámetros utilizando la técnica de aceleración de Nesterov, que predice los gradientes futuros y ajusta los parámetros en consecuencia, lo que puede acelerar la convergencia y mejorar la precisión.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \eta \left(\frac{\beta_1 \hat{m}_t}{1-\beta_1^t} + \frac{(1-\beta_1)\nabla_\theta J(\theta_t)}{1-\beta_1^t} \right)
\end{aligned}
$$

- **RAdam** [23]: Es una variante de Adam que introduce una rectificación adaptativa para manejar la variabilidad de los gradientes en las primeras etapas del entrenamiento. RAdam ajusta dinámicamente los pasos de actualización para rectificar la varianza del gradiente, proporcionando una mayor estabilidad durante las etapas iniciales del entrenamiento y mejorando la convergencia general.

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta_t) \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta_t))^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\rho_t &= \rho_\infty - 2t\frac{\beta_2^t}{1-\beta_2^t} \\
\rho_\infty &= \frac{2}{1-\beta_2} - 1 \\
r_t &= \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}} \\
\theta_{t+1} &= \begin{cases} \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} & \text{if } \rho_t > 4 \\ \theta_t - \eta \hat{m}_t & \text{otherwise} \end{cases}
\end{aligned}
$$

- **RProp** [29]: Es un optimizador que se centra en la dirección de los gradientes en lugar de su magnitud. Este enfoque permite que RProp sea menos sensible a la escala de los gradientes.

$$
\begin{aligned}
\Delta_{i,t} &= \begin{cases} \eta^+ \Delta_{i,t-1} & \text{if } \nabla_\theta J(\theta_t)\nabla_\theta J(\theta_{t-1}) > 0 \\ \eta^- \Delta_{i,t-1} & \text{if } \nabla_\theta J(\theta_t)\nabla_\theta J(\theta_{t-1}) < 0 \\ \Delta_{i,t-1} & \text{otherwise} \end{cases} \\
\theta_{i,t+1} &= \begin{cases} \theta_{i,t} - \Delta_{i,t} & \text{if } \nabla_\theta J(\theta_t) > 0 \\ \theta_{i,t} + \Delta_{i,t} & \text{if } \nabla_\theta J(\theta_t) < 0 \end{cases}
\end{aligned}
$$

- **LBFGS** [4]: Es un optimizador quasi-Newton que utiliza una aproximación limitada de la matriz Hessiana, lo que permite un uso eficiente de la memoria. LBFGS estima la matriz Hessiana de manera eficiente sin almacenarla explícitamente, utilizando información de los gradientes y actualizaciones pasadas. Esto permite que el optimizador maneje problemas de gran escala con alta eficiencia.

$$\theta_{t+1} = \theta_t - H_t^{-1} \nabla_\theta J(\theta_t)$$

A.2. Loss Functions de *Flogo*

- **MAE (Mean Absolute Error)** [36]: Esta función de pérdida mide el error promedio entre las predicciones y los valores verdaderos, tomando la media de las diferencias absolutas. Utilizado principalmente en problemas de regresión donde la interpretación directa de los errores en las unidades originales es importante.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$$

- **MSE (Mean Squared Error)** [36]: Mide el error promedio al cuadrado entre las predicciones y los valores verdaderos, tomando la media de los cuadrados de las diferencias. Comúnmente utilizado en problemas de regresión. Penaliza más los errores grandes debido al cuadrado de las diferencias.

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

- **Cross Entropy Loss** [36]: Mide la diferencia entre dos distribuciones de probabilidad: la distribución real y la distribución predicha. Utilizado en problemas de clasificación multicategoría.

$$\text{Cross Entropy Loss} = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

- **Binary Cross Entropy Loss** [36]: Es una variante de la entropía cruzada para problemas de clasificación binaria. Usado en problemas de clasificación binaria, donde las salidas son 0 o 1.

$$\text{Binary Cross Entropy Loss} = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

- **Huber Loss** [36]: Combina las ventajas de MAE y MSE, siendo menos sensible a outliers que MSE pero más robusto que MAE. Usado en problemas de regresión, especialmente cuando hay outliers presentes.

$$\text{Huber Loss} = \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{for } |y_i - \hat{y}_i| \leq \delta \\ \delta |y_i - \hat{y}_i| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

- **Kullback Leibler Divergence Loss** [36]: Mide cuánto una distribución de probabilidad se desvía de una segunda distribución de probabilidad esperada. Utilizado en problemas donde se desea medir la diferencia entre dos distribuciones de probabilidad, como en modelos de variational autoencoders.

$$\text{KL Divergence} = \sum_{i=1}^{n} y_i \log\left(\frac{y_i}{\hat{y}_i}\right)$$

- **Hinge Embedding Loss** [36]: Es una función de pérdida utilizada especialmente en tareas de clasificación binaria y aprendizaje de métricas. Esta pérdida es útil para medir la similitud o disimilitud entre pares de ejemplos en espacios de características, lo que la hace ideal para problemas donde se necesita aprender una métrica de distancia.

$$\text{Hinge Embedding Loss} = \begin{cases} y_i \operatorname{máx}(0, 1 - \hat{y}_i) & \text{if } y_i = 1 \\ (1 - y_i)\hat{y}_i & \text{if } y_i = -1 \end{cases}$$

- **CTC (Connectionist Temporal Classification Loss)** [36]: Diseñada para tareas de secuencias donde la alineación entre las entradas y las salidas no está disponible, como en el reconocimiento de voz. Usado en problemas de reconocimiento de voz, OCR y otras tareas de secuencias sin alineación.

$$\text{CTC Loss} = -\log(p(y|x))$$

- **MarginRankingLoss** [36]: Es una función de pérdida utilizada en problemas de ranking y aprendizaje de métricas. Evalúa la ordenación relativa entre pares de ejemplos y ajusta los pesos del modelo para asegurar que las puntuaciones de los pares se ordenen correctamente con respecto a un margen.

$$\text{MarginRankingLoss} = \operatorname{máx}(0, -y(x_1 - x_2) + \text{margin})$$

- **Triplet Margin Loss** [36]: Es una función de pérdida utilizada en el aprendizaje de métricas, particularmente en problemas donde se desea aprender una representación embebida de los datos. Se basa en la comparación de tríos de ejemplos.

$$\text{Triplet Margin Loss} = \operatorname{máx}(0, \|f(a) - f(p)\|_2^2 - \|f(a) - f(n)\|_2^2 + \text{margin})$$

- **Triplet Margin Loss With Swap** [36]: Una variante de Triplet Margin Loss que considera tanto la distancia entre el ancla y el positivo como entre el ancla y el negativo, y la distancia intercambiada. Mejora la robustez en el aprendizaje de métricas y es útil en tareas donde la precisión de la distancia relativa es crítica, como en la verificación de identidad y reconocimiento de objetos.

$$\begin{aligned}\text{Triplet Margin Loss With Swap} = \operatorname{máx}(0, &\|f(a) - f(p)\|_2^2 \\ &- \operatorname{mín}(\|f(a) - f(n)\|_2^2, \|f(p) - f(n)\|_2^2) + \text{margin})\end{aligned}$$

Apéndice B

Capas de la arquitectura

B.1. Capas de procesamiento de *Flogo*

- **Layer Normalization**: Esta capa normaliza los datos a lo largo de una dimensión específica, generalmente la última, asegurando que la entrada a cada capa tenga una media de cero y una desviación estándar de uno.
- **Batch Normalization**: Similar a la normalización por capas pero se aplica a través de todo el batch durante la inferencia o entrenamiento, es decir, normaliza los datos entre diferentes batches, ayudando a mejorar la estabilidad y el rendimiento del entrenamiento.
- **Dropout**: Utilizada como una técnica de regularización durante el entrenamiento para prevenir el sobreajuste. Esta capa funciona exactamente igual a una lineal pero aleatoriamente inhibe un porcentaje de las neuronas en cada paso de entrenamiento, lo que ayuda a prevenir el overfitting.
- **Linear**: Esta capa realiza una transformación lineal a los datos entrantes, $y = Wx + b$, donde W es la matriz de pesos y b es el vector de sesgos. La salida de esta capa es el vector y que tendrá de dimensión una de las dimensiones de W. La salida de estas capas se define mediante el constructo Output especificando la salida final de la capa.
- **Convolutional**: son esenciales para procesar imágenes y datos espaciales o temporales, utilizando kernels para extraer características importantes. Pueden configurarse especificando las dimensiones del kernel, como tamaño, stride, padding y número de canales de salida, o mediante el constructo Output que incluye largo, ancho y número de canales de salida del tensor resultante.
- **Pool**: También operan mediante el uso «kernel» con el objetivo de reducir la dimensionalidad de los datos, preservando las características más esenciales, estas aplican el kernel en subregiones para reducirlo a un único valor. Existen dos tipos principales: maxPooling, que selecciona el máximo de cada subregión, y averagePooling, que calcula el promedio de cada subregión. Estas capas pueden configurarse especificando el tamaño del kernel, el stride, y el padding, o definiendo directamente el Output, estableciendo

el largo y ancho de la salida.

- **Recurrent**: Estas capas son cruciales para el manejo de datos secuenciales, como series temporales o texto. En *Flogo* la estructura de estas capas están inspiradas en la metodología **map-reduce**, utilizada en el procesamiento de datos y programación paralela. En la operación de **mapeo**, estas capas utilizan unidades recurrentes, tales como **LSTM** (Long Short-Term Memory), **GRU** (Gated Recurrent Units), o **RNN simples** (Recurrent Neural Network). Cada una de estas unidades puede ser configurada con sus parámetros:
  - **NumLayers**: Este parámetro especifica el número de capas recurrentes apiladas en la red. Al aumentar el número de capas, la red puede aprender representaciones más complejas, pero también se incrementa la carga computacional y el riesgo de sobreajuste.
  - **Bidirectional**: Una red recurrente es bidireccional cuando procesa los datos tanto en la dirección desde el principio hasta el final de la secuencia como viceversa, aumentado en dos el número de capas. Esto permite que la red tenga en cuenta el contexto de ambos lados de un punto de datos, lo cual es especialmente útil en tareas donde la secuencia completa de datos es relevante.
  - **Dropout**: Es una técnica de regularización utilizada para prevenir el sobreajuste en las redes neuronales, especialmente en redes profundas. Al aplicar dropout, se "desactivan" aleatoriamente ciertas neuronas durante el entrenamiento, lo que hace que la red sea menos sensible a la especificidad de los datos de entrenamiento.
  - **Output**: El hidden size, con este parámetro se especifica la salida de cada capa o el tamaño de los estados ocultos en la unidad recurrente. Este tamaño es crucial porque determina la dimensión de la representación vectorial para cada paso de tiempo.

  En *Flogo*, las unidades recurrentes ofrecen una gama de salidas para que los desarrolladores seleccionen la más adecuada para sus necesidades específicas. Éstas son:
  - **Last Sequence**: Esto representa la salida de la última unidad recurrente en la secuencia de tiempo final. Esta salida es típicamente utilizada cuando solo nos interesa la predicción al final de la secuencia completa, como podría ser en tareas de clasificación de secuencias.
  - **Hidden States**: Se refiere a la colección de todos los estados ocultos generados por las unidades recurrentes a lo largo de toda la secuencia. Esto es útil en tareas donde cada momento de la secuencia es relevante, como en etiquetado de secuencias o cuando la secuencia completa se necesita para futuras operaciones, como la atención.
  - **Last Hidden State**: Esta es la salida del último estado oculto de la última capa recurrente. Es importante en modelos donde la representación condensada de toda la secuencia de entrada es suficiente para la tarea en cuestión, como podría ser la clasificaciónd de secuencias.
  - **Cell States** (específico de LSTM): Devuelve todos los estados de celda generados,

proporcionando una vista más completa de la memoria de la red a lo largo del tiempo.

- **Last Cell State** (específico de LSTM): Devuelve el último estado de celda, útil cuando solo el estado final de la memoria largo plazo es requerido.

En la ilustración B.1 se puede observar que tenemos una unidad LSTM que tiene de tamaño de secuencia seis y cuatro capas. Para cada salida, podremos ver cuál es el tamaño resultante.

Además, se introduce el concepto **from to**, en el que si la salida elegida es de tipo secuencia, se permite especificar desde que índice hasta que índice se desea extraer, facilitando así un ajuste más granular de los datos de salida.

Finalmente, la operación de **reduce** se puede realizar mediante dos métodos:

- **Linear**: Transforma la dimensión relacionada con el tamaño de la secuencia y la reduce a una sola, concentrando la información temporal en un vector fijo.
- **Flatten**: Realiza un aplanamiento (flat map) de las secuencias, permitiendo que las capas subsecuentes, que operan con datos de una dimensión, procesen los datos más eficientemente.

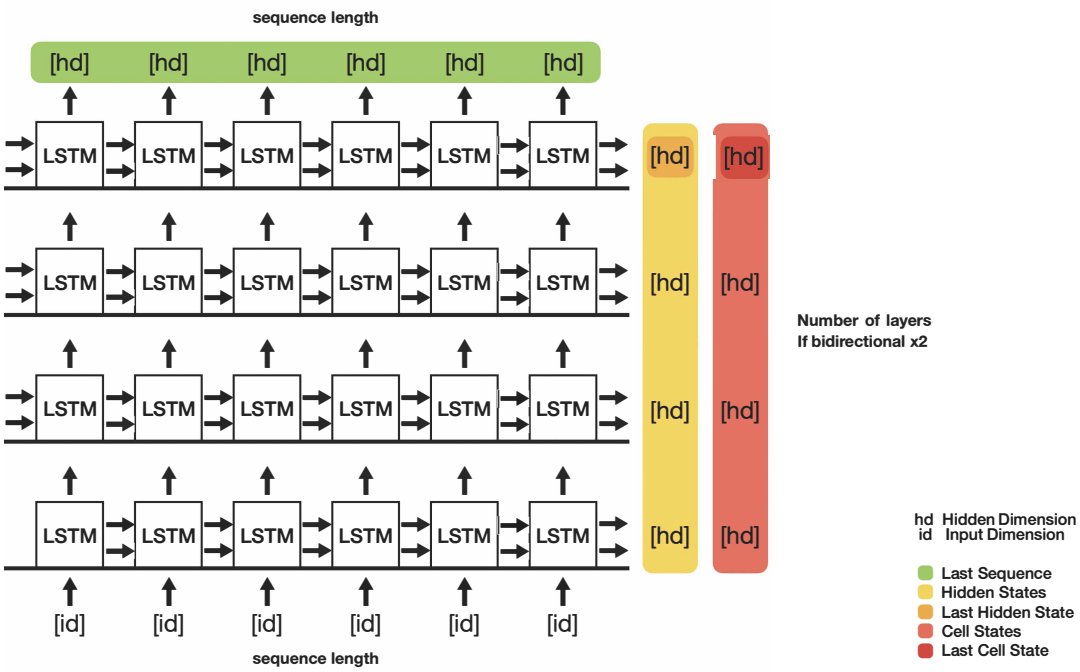

Ilustración B.1: Visualización de una capa LSTM en el *DSL Flogo*

B.2. Capas de activación de *Flogo*

- **Sigmoid** [11]: Convierte los valores de entrada en un rango entre 0 y 1, lo que la hace ideal para problemas de clasificación binaria, actuando como una función de probabilidad.

$$\text{Sigmoid}(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

- **ReLU (Rectified Linear Unit)** [11]: Proporciona una salida lineal para todos los valores positivos y cero para los valores negativos. Es muy utilizada debido a su simplicidad y eficiencia.

$$\text{ReLU}(x) = \text{máx}(0, x)$$

- **Softmax** [11]: Utilizada principalmente como última de las arquitecturas para clasificación multiclase, convierte las puntuaciones de las clases en probabilidades.

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

- **Tanh (Tangente Hiperbólica)** [11]: Transforma los valores de entrada en un rango entre -1 y 1. Es útil para modelar datos centrados alrededor de cero.

$$\text{Tanh}(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- **LeakyReLU** [11]: Una variante de ReLU que permite una pequeña pendiente, denotada normalmente como alpha, para valores negativos, evitando así el problema de neuronas inactivas en ReLU estándar.

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases}$$

- **LogSigmoid** [11]: Una variante de la función sigmoidal que aplica el logaritmo a la salida de la función sigmoid. Esto puede ayudar a estabilizar el aprendizaje evitando valores extremadamente pequeños o grandes en las salidas.

$$\text{LogSigmoid}(x) = \log(\sigma(x)) = \log\left(\frac{1}{1 + e^{-x}}\right)$$

- **Mish** [11]: Una función que ofrece beneficios similares a la ReLU pero con un rango de salida más suave que puede mejorar el aprendizaje.

$$\text{Mish}(x) = x \tanh(\log(1 + e^{x}))$$

- **SELU (Scaled Exponential Linear Unit)** [11]: Una versión escalada de ELU que auto-normaliza las neuronas, llevando a que cada capa mantenga una media y varianza cercanas a cero y uno, respectivamente.

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

- **GELU (Gaussian Error Linear Unit)** [11]: Multiplica la entrada por el valor de una función de distribución acumulativa de una distribución gaussiana.

$$\text{GELU}(x) = x \cdot \Phi(x)$$
$$\Phi(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

- **SiLU (Sigmoid Linear Unit)** [11]: También conocida como Swish, es el producto de la entrada por la función sigmoid de la entrada. Ha mostrado ser una alternativa eficaz a ReLU.

$$\text{SiLU}(x) = x \cdot \sigma(x) = x \cdot \frac{1}{1 + e^{-x}}$$

- **GLU (Gated Linear Unit)** [11]: Utiliza una puerta aprendida para controlar qué información pasará a través de la función de activación, combinando linealidad y capacidad de control.

$$\text{GLU}(a, b) = a \cdot \sigma(b)$$

- **ELU (Exponential Linear Unit)** [11]: Similar a ReLU, pero en lugar de ser cero para entradas negativas, la salida es una función exponencial en donde se denota la variable dependiente como alpha. Esto ayuda a reducir el problema del desvanecimiento de gradientes.

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$

Bibliografía

[1] Apache (2022). Apache mxnet. https://mxnet.apache.org/versions/1.9.1/. Consultado el 10 de febrero de 2024.

[2] Bebis, G. and Georgiopoulos, M. (1994). Feed-forward neural networks.

[3] Bettini, L. (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, Birmingham.

[4] Bollapragada, R., Mudigere, D., Nocedal, J., and et. al. (2018). A progressive batching l-bfgs method for machine learning. *arXiv*.

[5] Brown, S. A., Drayton, E., and Mittman (1963). A description of the apt language. *ACM*.

[6] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., and et al (2020). Language models are few-shot learners.

[7] Campagne, F. (2014). *The MPS language workbench: volume I*, volume 1. Fabien Campagne, New York.

[8] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database.

[9] Deng, L. (2012). The mnist database of handwritten digit images for machine learning research.

[10] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding.

[11] Dubey, S. R., Singh, S. K., and Chaudhuri, B. B. (2022). Activation functions in deep learning: A comprehensive survey and benchmark. *arXiv*.

[12] F., R. (1958). The perceptron: A probabilistic model for information storage and organization in the brain.

[13] Fowler, M. (2019). Domain-specific languages guide. https://www.martinfowler.com/dsl.html. Consultado el 7 de enero de 2024.

[14] Frederick P. Brooks, J. (1975). *Mythical Man-Month*. Addison Wesley, 1 edition.

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

[16] GoogleBrainTeam (2015). Tensorflow. https://www.tensorflow.org/. Consultado el 10 de febrero de 2024.

[17] Gálvez, J. J. H. (2024). Neuralflogo/dsl. https://github.com/NeuralFlogo/dsl. Consultado el 10 de junio de 2024.

[18] Hotegni, S. S., Berkemeier, M., and Peitz, S. (2024). Multi-objective optimization for sparse deep multi-task learning. *arXiv*.

[19] I, J. M. (1986). Serial order: a parallel distributed processing approach.

[20] Jumper, Evans, Pritzel, Green, Figurnov, Ronneberger, and et al (2021). Highly accurate protein structure prediction with alphafold.

[21] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks.

[22] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning.

[23] Liu, L., Jiang, H., He, P., and et. al. (2019). On the variance of the adaptive learning rate and beyond. *arXiv*.

[24] Loshchilov, I. and Hutter, F. (2019). Decoupled weight decay regularization. *arXiv*.

[25] Martin, R. C. (2008). *Clean Code*. Pearson.

[26] Maxfield, M. (2018). High level shading language. https://www.ibm.com/docs/es/iis/11.7?topic=statement-backus-naur-form-bnf-notation. Consultado el 10 de febrero de 2024.

[27] McCulloh, W. S. and Pitts, W. (1943). A logical calculus of the ideas immament in nervous activity.

[28] Microsoft (2017). Microsoft cognitive toolkit. https://github.com/microsoft/CNTK. Consultado el 10 de febrero de 2024.

[29] Mosca, A., Magoulas, G. D., and et. al. (2015). Adapting resilient propagation for deep learning. *arXiv*.

[30] Paszke, A., Gross, S., Chintala, S., and Chanan, G. (2016). Pytorch. https://pytorch.org/. Consultado el 10 de febrero de 2024.

[31] Rajan, H. (2022). *Appendix: ANTLR: A Brief Review*, pages 265–268. MIT Press, Cambridge, Massachusetts.

[32] Ramírez, M. C., Andrés, O. R., Cabrera, J. J. H., and Évora, J. (2016). Itrules. https://plugins.jetbrains.com/plugin/7748-itrules. Consultado el 2 de marzo de 2024.

[33] Reddi, S. J., Kale, S., and Kumar, S. (2019). On the convergence of adam and beyond. *arXiv*.

[34] Ruder, S. (2017). An overview of gradient descent optimization algorithms. *arXiv*.

[35] Rumelhart, Hinton, and Williams (1986). Learning representations by back-propagating errors.

[36] Terven, J., Cordova-Esparza, D. M., Ramirez-Pedraza, A., and Chavez-Urbiola, E. A. (2023). Loss functions and metrics in deep learning. *arXiv*.

[37] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2018). Attention is all you need.

[38] Zheng, S., Meng, Q., Wang, T., and et al (2020). Asynchronous stochastic gradient descent with delay compensation. *arXiv*.

[39] Zhuang, J., Ding, Y., Tang, T., Dvornek, N., Tatikonda1, S., and Duncan1, J. S. (2021). Momentum centering and asynchronous update for adaptive gradient methods. *NeurIPS*.

Glosario

AI Winter período en la historia de la inteligencia artificial durante en el cual la inversión y el interés en la investigación de IA disminuyeron significativamente debido a resultados decepcionantes.. 9

AlphaFold modelo desarrollado por DeepMind que es capaz de predecir con bastante acierto la estructura tridimensional de las proteínas a partir de sus secuencias de aminoácidos.. 9

API REST interfaz de programación de aplicaciones que sigue los principios de la transferencia de estado representacional (REST), utilizando métodos HTTP para interactuar con recursos web. 32

Back Propagation algoritmo utilizado en redes neuronales para ajustar los pesos de las conexiones en función del error entre la salida real y la salida deseada, optimizando así el rendimiento del modelo.. 9

BERT *Bidirectional Encoder Representations from Transformers*, un modelo de lenguaje desarrollado por *Google* que se entrena bidireccionalmente para entender el contexto de las palabras en una oración.. 9

Check Point Saver mecanismo que guarda los parámetros del modelo en ciertos puntos durante el entrenamiento, típicamente cuando el modelo alcanza una nueva mejor performance en el conjunto de validación. Esto permite recuperar el mejor modelo en caso de interrupción del entrenamiento. 15

Dataset colección de datos utilizada para entrenar y evaluar modelos de aprendizaje automático. Un dataset generalmente se divide en subconjuntos de entrenamiento, validación y prueba.. 15

Deep Learning subcampo de la inteligencia artificial que utiliza redes neuronales artificiales con muchas capas (redes profundas) para modelar y entender datos complejos.. 4, 6, 8, 80

Deep Neural Networks redes neuronales con múltiples capas entre la capa de entrada y la capa de salida, que permiten modelar y aprender representaciones complejas de datos.. 9

Early Stopper Técnica utilizada durante el entrenamiento de un modelo para detener el

proceso si la exactitud en el conjunto de validación no mejora después de un número determinado de épocas. Esto ayuda a prevenir el sobreajuste. 15

Fine Tuning Un proceso en el aprendizaje automático donde un modelo preentrenado se ajusta específicamente para una tarea particular mediante un entrenamiento adicional con un conjunto de datos más pequeño y específico. El fine tuning permite mejorar la precisión del modelo para la tarea objetivo aprovechando el conocimiento previamente adquirido.. 61

Framework conjunto estructurado de prácticas y herramientas que proveen una base conceptual para el desarrollo de software. Estos facilitan la creación y el mantenimiento de aplicaciones complejas al ofrecer componentes reutilizables y soluciones predefinidas.. 1

Función de Onda mecánica cuántica, una función matemática que describe el estado cuántico de una partícula o sistema de partículas.. 19

Física Cuántica rama de la física que estudia los fenómenos a escalas nanoscópicas, donde las leyes de la mecánica cuántica predominan sobre las de la mecánica clásica.. 19

GET método HTTP utilizado para solicitar datos de un recurso web específico.. 32

GPT-3 *Generative Pre-trained Transformer 3*, un modelo de lenguaje desarrollado por *OpenAI* que utiliza aprendizaje profundo para producir texto similar al humano.. 9

IDE Entorno de Desarrollo Integrado (Integrated Development Environment), un software que proporciona herramientas completas para el desarrollo de software, como el editor de código, el depurador.... 50

Internship práctica profesional temporal que permite a los estudiantes o recién graduados obtener experiencia laboral en su campo de estudio.. 44

JSON *JavaScript Object Notation*, formato de intercambio de datos ligero y fácil de leer/escribir tanto para humanos como para máquinas.. 32

Kernel en el contexto de la inteligencia artificial y específicamente en redes neuronales convolucionales, el término "kernel" se refiere a una pequeña matriz de pesos utilizada para aplicar convoluciones a una imagen o a otra entrada de datos. Los kernels se deslizan sobre la entrada para extraer características importantes, como bordes, texturas y patrones, facilitando la identificación de características relevantes en tareas de visión por computadora.. 23

lenguajes de propósito general lenguajes de programación diseñados para ser utilizados en una amplia variedad de aplicaciones y dominios, como Python, Java, y C++.. 3

lenguajes específicos de dominio lenguajes de programación diseñados para resolver problemas en un dominio particular o un área específica, como SQL para bases de datos o HTML para el marcado de documentos web.. 3

Loss Function función que mide la diferencia entre las predicciones del modelo y los valores reales. Se utiliza durante el entrenamiento del modelo para guiar la optimización.. 15

Map-Reduce modelo de programación para procesar grandes conjuntos de datos de manera paralela y distribuida. Consiste en dos fases: **map**, que hace alguna operación sobre los datos, y **reduce**, que consolida estos datos transformados en un resultado más compacto.. 23

Markdown lenguaje de marcado ligero que se utiliza para formatear texto en la web, facilitando la creación de documentos con formato sencillo y fácil de leer.. 57

Model Driven Engineering enfoque de ingeniería de software que utiliza modelos abstractos para dirigir y automatizar el desarrollo del software.. 24

Optimizer algoritmo o método utilizado para ajustar los parámetros de un modelo de aprendizaje automático con el fin de minimizar o maximizar una función objetivo, como la función de pérdida.. 15

POST método HTTP utilizado para enviar datos al servidor para crear o actualizar un recurso.. 32

Principio de sustitución de Liskov principio de la programación orientada a objetos que pertenece a los principios SOLID y establece que los objetos de una clase derivada deben poder sustituir a los objetos de su clase base sin alterar el funcionamiento correcto del programa. 6

Proteo lenguaje específico de dominio (*DSL*), especializado en el modelado de conceptos. Proteo se utiliza para definir *DSLs* y facilitar la configuración y ejecución de experimentos en diversos entornos. 25

ResNet abreviatura de Red Residual, es un tipo de red neuronal convolucional que utiliza *Shortcuts* para saltar algunas capas, permitiendo el entrenamiento de redes mucho más profundas al abordar el problema de desvanecimiento de gradiente. . 42

Serverless modelo de computación en la nube donde el proveedor se encarga de la gestión de los servidores, permitiendo a los desarrolladores centrarse en el código sin preocuparse por la infraestructura subyacente.. 34

Strategy método o enfoque utilizado para entrenar un modelo de *Deep Learning*. En *Flogo* hay dos tipos de regresión y de clasificación. 15

Toy Problem problema simplificado y bien definido que permite a los investigadores probar y perfeccionar sus modelos en un entorno controlado.. 34

Transfer Learning Una técnica en el aprendizaje automático donde un modelo desarrollado para una tarea inicial se reutiliza como punto de partida para una tarea diferente pero relacionada. El transfer learning aprovecha el conocimiento adquirido en la tarea inicial para mejorar el rendimiento y reducir el tiempo de entrenamiento en la nueva tarea.. 61