

Framework para el Aprendizaje de Redes Neuronales Profundas

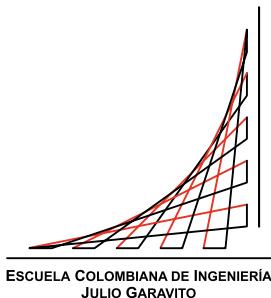
*Proyecto Dirigido
Ingeniería de Sistemas*

Autores:

Alejandro Anzola Ávila
Juan Andrés Moreno Silva

Directora:

María Irma Díaz Rozo



Decanatura de Ingeniería de Sistemas
Escuela Colombiana de Ingeniería Julio Garavito
Bogotá, Colombia
Julio 2020



Resumen

El aprendizaje automático es un campo de la inteligencia artificial que se ha desarrollado exponencialmente en los últimos años gracias a la inmensa cantidad de datos disponibles, al enorme poder de procesamiento con el que ahora se cuenta y a las innovaciones tecnológicas que se han logrado en este campo. Dentro de las grandes fortalezas que ofrece esta tecnología es la de poder resolver una amplia cantidad de problemas de diferentes dominios y áreas del conocimiento, e incluso es posible reusar el conocimiento y el modelo obtenido al resolver un problema en problemas de dominios similares.

Uno de los principales actores en esta revolución son las redes neuronales, que pertenecen al campo del aprendizaje automático. Esta tecnología nos permite resolver una infinidad de problemas, desde clasificar números hasta generar código dado una instrucción en lenguaje natural.

Sin embargo, esta tecnología tiene una curva de aprendizaje elevada. Nuestra solución a este problema es ofrecer recursos de aprendizaje de redes neuronales para que el estudiante pueda aprender conceptos bases a través de la experimentación y la extensión de los mismos. Los recursos desarrollados son (i) un *framework* de aprendizaje (marco de trabajo) y (ii) un conjunto de casos problema-solución.

El *framework* (marco de trabajo) permite que los estudiantes puedan construir y probar modelos de redes neuronales. Este *framework* está diseñado con el fin de ser una ayuda para la comprensión de conceptos claves a novatos interesadas en el campo, siguiendo lineamientos de buenas prácticas de ingeniería de software.

El *framework* está pensado para representar los conceptos base de inteligencia artificial, de manera suficientemente general como para poder extenderlo para implementar diferentes modelos de redes neuronales. Para esto, el *framework* cuenta con distintos componentes extensibles que corresponden a los principales conceptos de redes neuronales. Dentro de estos componentes se encuentran: los tipos de redes neuronales (profundas, recurrentes y convolucionales); las funciones de activación y costo; los mecanismos de regularización y optimización, entre otros.

En el conjunto de casos problema-solución se trabajaron tres problemas clásicos y un problema especializado en el área de hidrología. Los problemas clásicos seleccionados fueron la clasificación de dígitos escritos a mano con redes densas y convolucionales y la clasificación de textos (positivos ó negativos) con redes recurrentes. El problema en hidrología fue la predicción de un caudal en el río Magdalena, en este trabajo se realizaron dos modelos con el fin de evaluarlas: un perceptrón multicapas (MLP por sus siglas en inglés (*MultiLayer Perceptron*) y un modelo secuencial de capas con memoria (LSTM por sus siglas en inglés (*Long-Short Term Memory*)) con el fin de evaluarlas.

Los casos desarrollados nos permitieron, además de generar recursos de aprendizaje, probar la versatilidad del *framework*. En tres de los cuatro casos se comprobó el desempeño adecuado del *framework*. El caso que no fue posible implementar fue el de redes neuronales recurrentes por problemas de alcance; sin embargo, están consideradas en el diseño de la estructura del *framework*.

En la implementación de los casos se evidenció que el *framework* trabaja alineado con la filosofía de buenas prácticas de desarrollo de software. En particular, en la implementación del problema en hidrología, se demostró que esta especialmente diseñado para tener fácil extensibilidad.

Una de las métricas de desempeño que se tomaron fueron los tiempos de entrenamiento, aunque no es parte de los objetivos, de forma que se tuviera una forma de comparación en ese aspecto. No fue inesperado encontrar que el desempeño en Keras superó considerablemente a la implementación hecha en el *framework*.

Finalmente, uno de los aspectos claves del *framework* es que fue diseñado con modelos supervisados en mente, lo cual permite extenderlo con modelos supervisados diferentes a las redes neuronales. Como trabajo futuro se desea incluir las líneas de aprendizaje no-supervisado ó aprendizaje reforzado.

Abstract

Machine Learning is a field of artificial intelligence that has been developing exponentially in the last few years due to the immense amount of data, processing power, and technology innovations. Among the great strengths that this technology offers is the vast amount of problems from different domains and knowledge fields. It is even possible to reuse knowledge and models obtained after solving a problem from a similar field.

One of the main actors in this revolution is the neural networks technology, which is part of the field of machine learning. This technology lets us solve an infinite amount of problems, from classifying written numbers to source code generation from natural language.

However, this technology has a very steep learning curve. Our solution to this problem is to offer neural network learning resources to help students learn base concepts through experimentation and extension. The developed resources are (i) a learning framework and (ii) a set of problem-solution cases.

The framework lets students build and test neural network models. This framework is designed to aid newcomers interested in the field to understand base concepts, following software engineering best practices.

This framework is built to represent artificial intelligence base concepts, in a way general enough to be extensible to different neural network models. To achieve this, the framework has various extensible components that correspond to different main concepts of neural networks. Among these components are (i) neural network types (deep, recurrent, and convolutional); (ii) activation and loss functions; and (iii) regularization and optimization mechanisms.

In the set of problem-solution cases, there are three classic problems and one specialized problem in the field of hydrology. The classic problems selected are handwritten digits with dense and convolutional networks, and positive-negative text classification with recurrent networks. The hydrology problem was the prediction of flow in the Magdalena river; in this work, we made two models to evaluate them: a multilayer perceptron (MLP for short) and a Long-short Term Memory (LSTM for short) for evaluation.

The cases developed allowed us to test the versatility of our framework in addition to having learning resources. In three of the four cases, the performance of the framework was verified and deemed appropriate. The case that could not be implemented was the one that required recurrent neural networks due to encountered scope problems; however, recurrent neural networks are considered in the framework structure's design.

In the implementation of the cases, it could be seen that the framework works aligned with the philosophy of good software development practices. In particular, in the implementation of the hydrology problem, it was shown that it is easy to extend.

One of the performance metrics that were taken was training times to have a way of comparing them, although it is not part of our objectives. It was not unexpected to find that Keras' performance is considerably superior to the implementation made in our framework.

Finally, one of our framework's key aspects is that it was designed with only supervised models in mind, which only allows it to be extended to supervised models other than neural networks. The addition of unsupervised learning or reinforced learning is future work.

Índice general

1. Introducción	2
1.1. Motivación	3
1.2. Enfoque propuesto	3
1.2.1. Fase I: <i>Framework</i> para el aprendizaje de redes neuronales profundas.	3
1.2.2. Fase II: <i>Framework</i> para el aprendizaje de redes neuronales profundas	
Caso modelos hidrológicos: El río Magdalena.	4
1.3. Resultados esperados	5
1.4. Contenido	6
2. Redes neuronales: una aproximación orientada a objetos	7
2.1. Introducción	8
2.2. Marco teórico	8
2.2.1. Redes Neuronales	8
2.2.2. Programación Orientada a Objetos	11
2.2.3. Código Limpio (<i>Clean Code</i>)	12
2.3. Trabajos Relacionados	12
2.4. Solución	15
2.4.1. Diseño	15
2.4.2. Implementación	34
2.4.3. Usabilidad	35
2.5. Evaluación	35
2.5.1. Plan	35
2.5.2. Caso: <i>MNIST</i>	36
2.5.3. Caso: Clasificación de textos	37
2.5.4. Hallazgos	38
2.6. Conclusiones	38
2.7. Trabajo futuro	39
3. Predicción del caudal Calamar del Río Magdalena con Redes Neuronales	41
3.1. Introducción	42
3.2. Marco teórico	43
3.2.1. Hidrología	43
3.2.2. Serie de tiempo	44
3.2.3. Ventana de promedio deslizante	45
3.2.4. Modelo ARIMA	46

3.2.5. Modelo TRAMO	47
3.2.6. Red neuronal artificial	48
3.3. Trabajos relacionados	50
3.3.1. Métodos estadísticos	50
3.3.2. Métodos con redes neuronales	51
3.4. Solución	51
3.4.1. Especificación del Problema	51
3.4.2. Descripción de datos	52
3.4.3. Estrategia de trabajo	57
3.4.4. Modelo MLP	58
3.4.5. Modelo LSTM	58
3.5. Evaluación	60
3.5.1. Resultados	60
3.5.2. Hallazgos	62
3.6. Conclusiones	62
3.7. Trabajo futuro	62
4. Predicción del caudal de Calamar del Río Magdalena utilizando un framework orientado a objetos	63
4.1. Introducción	64
4.2. Marco teórico	65
4.2.1. <i>Framework</i>	65
4.2.2. Modelos hidrológicos	66
4.3. Trabajos relacionados	66
4.3.1. Frameworks orientados a objetos	66
4.3.2. Modelos hidrológicos con redes neuronales	67
4.4. Solución	67
4.4.1. Especificación del Problema	67
4.4.2. Estrategia de trabajo	67
4.4.3. Diseño	68
4.4.4. Implementación de los modelos en los respectivos <i>frameworks</i>	68
4.5. Evaluación	71
4.5.1. Resultados	71
4.5.2. Hallazgos	74
4.6. Conclusiones	74
4.7. Trabajo futuro	74
A. Notación	75
Índice de figuras	78
Índice de cuadros	80
Referencias	

Agradecimientos

Queremos agradecer a todas las personas que estuvieron a lo largo de nuestra carrera de pregrado.

Queremos empezar agradeciendo muy especialmente al Dr. Oswaldo Castillo Navetty por su gran apoyo durante toda nuestra carrera. Sus valiosos consejos, su incondicional apoyo y su amistad fueron una guía y ayuda invaluable que muy pocas personas están dispuestas a dar.

También queremos agradecer de manera muy especial a nuestra directora de proyecto María Irma Díaz Rozo, quien durante nuestra carrera ha sido nuestra luz en el camino de la Ingeniería de Software y la Inteligencia Artificial, sin sus valiosos conocimientos, su guía y apoyo no seríamos los profesionales que somos hoy en día.

A nuestras familias y especialmente a nuestras madres. Sin su amor, apoyo y dedicación no hubiéramos podido llegar a ser las personas que somos.

A la profesora Claudia Patricia Santiago Cely, a Nubia Aurora León Duarte y a Alba Mariela Barbosa Triviño por sus consejos y enseñanzas durante nuestras monitorías del laboratorio de informática.

A Héctor Fabio Cadavid Rengifo quien con sus valiosas recomendaciones nos ayudó a desarrollar proyectos de software para la universidad.

A Wilmer Garzón Alfonso por su valioso esfuerzo y apoyo en las competencias del Colombian Collegiate Programming League (CCPL).

Queremos también dar gracias a nuestros compañeros que durante la carrera nos dieron buenos consejos, nos apoyaron en situaciones personales, y nos acompañaron en diversas competencias.

Sin duda este trabajo no se hubiera podido realizar sin ayuda de ustedes, por este motivo se los dedicamos.

Capítulo 1

Introducción

1.1. Motivación

Las redes neuronales profundas (**DNN** Deep Neural Networks) se han posicionado como una de las tecnologías más exitosas para la implementación de cierto tipo de soluciones inteligentes. Este campo de la inteligencia artificial está actualmente en pleno desarrollo gracias a los resultados obtenidos en los proyectos de investigación y de innovación en el área.

Para usar esta tecnología de forma adecuada, los profesionales en informática deben conocerla en profundidad tanto en la teoría (supuestos, conceptos, bases científicas y algoritmos asociados) como en la práctica (diseño, entrenamiento, sintonización, selección, verificación y validación de soluciones). El ambiente de aprendizaje propicio para tecnologías como ésta es el aprendizaje auténtico: aprender haciendo lo que hacen los profesionales cuando se enfrentan a casos reales.

El propósito de este proyecto es construir recursos de aprendizaje auténtico para el área de redes neuronales profundas, específicamente:

1. Un *framework* interactivo de aprendizaje y experimentación para **DNN**.
2. Soluciones a problemas clásicos de **DNN**
3. Soluciones a problemas especializados de **DNN**: modelos hidrológicos.

La motivación para la selección del tema para los problemas especializados, modelos hidrológicos, fue el concurso “Conéctate con el Río Magdalena” organizado por la Escuela Colombiana de Ingeniería Julio Garavito.

1.2. Enfoque propuesto

El proyecto se realizará en dos fases. En la primera fase se desarrollará la versión inicial del *framework* y se implementarán tres soluciones a problemas clásicos de **DNN**; y en la segunda fase, se perfeccionará el *framework* y se solucionará un problema real en el área de hidrología.

A continuación se detallan los objetivos específicos y la metodología de cada una de las fases.

1.2.1. Fase I: *Framework* para el aprendizaje de redes neuronales profundas.

Período 2020-1 2020-I

1.2.1.1. Objetivos

El objetivo general de este trabajo es desarrollar un *framework* diseñado específicamente para ser una herramienta interactiva de aprendizaje y experimentación para las personas interesadas en las redes neuronales dentro del campo del aprendizaje profundo. Los objetivos específicos son:

- Desarrollar un *framework* para redes neuronales profundas.
- Implementar tres soluciones que ilustren su uso. Los problemas seleccionados deben permitir probar las redes clásicas de aprendizaje profundo: FNN (Full Neural Network), CNN (Convolutional Neural Network) y RNN (Recurrent Neural Network).

El diseño e implementación se realizarán siguiendo estándares y buenas prácticas necesarios para cumplir con los requisitos de claridad, legibilidad y extensibilidad.

1.2.1.2. Metodología

El procedimiento propuesto para lograr los objetivos contempla una revisión de marco teórico y el uso de dos metodologías: una para el desarrollo del *framework* y otra para el desarrollo de las redes.

La metodología para el **desarrollo del framework** incluye las fases definidas en UP (Unified Process) para desarrollo de software: a) inicio, b) elaboración, c) construcción y d) transición. La construcción se realizará considerando cuatro ciclos de desarrollo: i) perceptrón multicapas, ii) mecanismos de optimización y regularización, iii) red convolucional, iv) red secuencial.

La metodología a seguir para el **desarrollo de redes neuronales** es la definida para desarrollo de modelos de aprendizaje automático: a) definición del problema, b) preprocesamiento de datos, c) entrenamiento de las redes candidatas, d) selección de la red, e) prueba de la red, f) despliegue de la red.

1.2.1.3. Secuencia y tipo de actividades

1. Revisión de marco teórico
2. Desarrollo del *framework*
 - 2.1 Inicio
 - 2.2 Elaboración
 - 2.3 Construcción
 - 2.3.1. Construcción ciclo uno: perceptrón multicapas
 - 2.3.2. Construcción ciclo dos: mecanismos de optimización y regularización
 - 2.3.3. Construcción ciclo tres: red convolucional
 - 2.3.4. Construcción ciclo cuatro: red recurrente
 - 2.4 Transición
3. Desarrollo de soluciones
 - 3.1 Solución FNN
 - 3.2 Solución CNN
 - 3.3 Solución RNN
4. Escritura del artículo

1.2.2. Fase II: *Framework* para el aprendizaje de redes neuronales profundas Caso modelos hidrológicos: El río Magdalena.

1.2.2.1. Objetivos

El objetivo general del proyecto es desarrollar recursos de aprendizaje auténticos para redes neuronales profundas. Los objetivos específicos asociados son:

- Aplicar la tecnología de redes neuronales profundas en la solución de problemas de hidrología
- Validar, refactorizar y extender el poder expresivo del *framework* para implementar las soluciones
- Construir los recursos de aprendizaje asociados a las soluciones

1.2.2.2. Metodología

El procedimiento propuesto para lograr los objetivos contempla una revisión de marco teórico, análisis y preprocesamiento inicial de datos, selección de pares problema-tecnología a trabajar, desarrollo de los

modelos hidrológicos en Keras y en el *framework*, perfeccionamiento del *framework* y despliegue de los recursos de aprendizaje.

Keras es una librería profesional, desarrollada para aprendizaje profundo, que se usará para desarrollar los modelos hidrológicos y validar la capacidad de nuestro *framework* para implementar esos modelos. Los modelos desarrollados en Keras se tomarán como línea base de las soluciones que se implementarán en el *framework* de aprendizaje.

La **revisión del marco teórico** es centrará en trabajos relacionados en el área de hidrología para comprender el área de dominio y para poder decidir los datos que son relevantes para los modelos.

Debido a que los datos del concurso “Conéctate con el río Magdalena” son datos públicos y tomados a lo largo de 30 años, muchos de estos datos están incompletos y contienen inconsistencias, por lo que uno de los primeros pasos es realizar un **análisis y preprocesamiento inicial del conjunto de datos**. Esto implica realizar una limpieza de los datos innecesarios, completar los datos que lo requieran y realizar normalización estadística para que el modelo pueda aprender correctamente con los datos.

Para la **selección de pares problema-tecnología**, el problema central será el predecir el caudal de un río – hay pocos modelos que logran una buena aproximación – y se considerarán aproximaciones diferentes de aprendizaje profundo: perceptrón multicapas, convolucionales y recurrentes.

El **desarrollo de los modelos hidrológicos en Keras y en el framework** para los pares problema-tecnología anteriormente definidos se implementarán primero en Keras y al tener modelos sintonizados en esta herramienta se pasará a hacer la implementación en el *framework* de aprendizaje. Adicionalmente, se realizará la comparación y análisis de los resultados de los modelos desarrollados en las dos herramientas.

El **perfeccionamiento del framework** se realizará para lograr que sea lo suficientemente expresivo como para realizar una implementación equivalente a la implementación profesional realizada en Keras.

La metodología a seguir para el **desarrollo de redes neuronales** es la definida para desarrollo de modelos de aprendizaje automático: a) definición del problema, b) preprocesamiento, c) entrenamiento de las redes candidatas, d) selección de la mejor red, e) prueba de la red seleccionada, y f) despliegue de la red

1.2.2.3. Secuencia y tipo de actividades

1. Revisión de marco teórico
2. Análisis y preprocesamiento de datos
3. Selección de pares problema-tecnología
4. Desarrollo de los modelos hidrológicos
 - 4.1 Desarrollo de los modelos en Keras
 - 4.1.1. Generación del modelo base
 - 4.1.2. Optimización y afinamiento
 - 4.2 Desarrollo de los modelos en el *framework*
 - 4.3 Comparación de resultados
5. Validación, refactorización y extensión del *framework*
6. Despliegue de los recursos de aprendizaje
7. Escritura de los artículos

1.3. Resultados esperados

Los resultados esperados de este proyecto, alineados con los objetivos, son los siguientes:

- Fase I. *Framework* para **DNN**
- Fase I. Tres soluciones a problemas clásicos en **DNN**
- Fase II. Soluciones a problemas especializados en el área de hidrología.

Adicionalmente, se escribirán tres artículos técnicos:

- Fase I. Redes neuronales: una aproximación orientada a objetos
- Fase II. Predicción del caudal Calamar del río Magdalena con Redes Neuronales
- Fase II. Predicción del caudal de Calamar utilizando un *framework* orientado a objetos

1.4. Contenido

El documento está estructurado en tres capítulos.

- Capítulo 2. Redes neuronales: una aproximación orientada a objetos
- Capítulo 3. Predicción del caudal Calamar del río Magdalena con redes neuronales
- Capítulo 4. Predicción del caudal de Calamar utilizando un *framework* orientado a objetos

Capítulo 2

Redes neuronales: una aproximación orientada a objetos

Resumen

En este artículo presentamos un *framework* de redes neuronales, diseñado específicamente para ser una herramienta interactiva de aprendizaje y experimentación para los interesados en aprender acerca de redes neuronales dentro del campo del aprendizaje profundo. El *framework* ilustra los detalles relevantes en el diseño de una arquitectura neuronal y es simple de usar. El diseño es suficientemente expresivo como para realizar una implementación de diferentes tipos de redes y tanto el diseño como la implementación siguen estándares y buenas prácticas de programación que proveen de legibilidad y claridad para ser fácilmente extensible. El *framework* fue probado con tres problemas clásicos de inteligencia artificial (Clasificación de caracteres con redes densas, clasificación de caracteres con redes convolucionales, clasificación de frases). Finalmente concluimos que el *framework* es lo suficiente expresivo y extensible para cubrir la mayoría de los modelos de aprendizaje profundo.

2.1. Introducción

En la última década hemos vivido un aumento importante en la cantidad de temas e investigaciones relacionadas con aprendizaje automático (ML *Machine Learning*) y aprendizaje profundo (DL *Deep Learning*). Uno de los principales resultados de este movimiento es que estamos experimentando una revolución digital con aplicaciones inteligentes desarrolladas en distintas áreas, como la detección facial, la clasificación de imágenes, entre otras.

Los problemas que se tenían antes, con esta tecnología, es que no se contaba con los billones de datos que hoy tenemos y que no se podía procesar grandes cantidades de datos por el alto costo que esto conllevaba. Afortunadamente no solo ha crecido la cantidad de datos, sino que a su vez la capacidad de computo ha crecido considerablemente. Ahora cualquier persona con conocimientos de ML y con una conexión a internet puede desarrollar modelos y analizar grandes cantidades de datos a bajo costo (en muchos casos gratis) y con un gran poder computacional.

Gran parte de este crecimiento se debe a que hay cientos de cursos de manera *online* y gratuita para que las personas puedan aprender sobre estas tecnologías. También, debido a esta revolución, se han ido creando comunidades enteras que buscan divulgar, enseñar y resumir gran parte de estas tecnologías en *post* y *blogs*.

El principal problema de estas publicaciones informales es que no dan todos los detalles relevantes, ó solo se centran en explicar los casos más triviales. Por otro lado el código de este tipo de publicaciones tiende a ser desordenado y poco extensible (código espagueti); en consecuencia, el código es poco legible, y dificulta la comprensión de los algoritmos.

En este artículo presentamos un *framework* de redes neuronales que permite ilustrar los conceptos de redes neuronales de una manera clara, completa y simple. El *framework* está hecho con el paradigma orientado a objetos y con buenas prácticas de programación.

2.2. Marco teórico

2.2.1. Redes Neuronales

Una *red neuronal artificial* (ANN *Artificial Neural Network*) es un modelo paramétrico de aprendizaje automático (ML) usada para aproximar la distribución de un conjunto de datos. Reciben este nombre ya

que se inspiran en la estructura de las redes neuronales del cerebro. Las redes neuronales logran esto al realizar una estimación inicial y realizar un ajuste de sus parámetros basado en una métrica de desempeño. La figura 2.1 ilustra la estructura de una red neuronal.

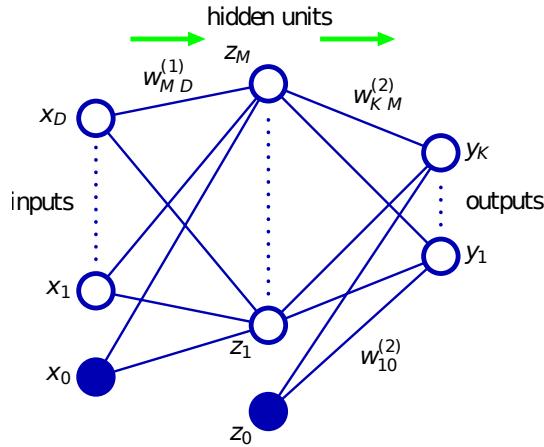


Figura 2.1: Red perceptrón multicapa (MLP). Tomado de Bishop (2006).

2.2.1.1. Aprendizaje profundo (*Deep Learning*)

Es un subconjunto de las ANN que se basa en diferentes técnicas de las redes neuronales. El concepto fundamental de DL (*Deep Learning*) se describe a continuación:

“Esta solución permite a los computadores a aprender de la experiencia y entender el mundo en términos de una jerarquía de conceptos, con cada concepto definido a través de su relación con conceptos más simples.” . . . “Si se dibuja un grafo mostrando como estos conceptos están construidos sobre otros, este grafo es profundo. Por esta razón, a esta aproximación se le llama aprendizaje profundo.”
— Adaptado de (Goodfellow *et al.*, 2016).

2.2.1.2. Aprendizaje automático (*Machine Learning*)

Se define como

“Un conjunto de métodos que puede detectar patrones en datos, y luego usarlos para predecir datos futuros, o realizar otro tipo de toma de decisiones bajo una incertidumbre”

— Adaptado de (Murphy, 2012).

2.2.1.3. Perceptrón

El perceptrón es uno de los componentes fundamentales en la mayoría de redes neuronales. Para entenderlo supóngase que tenemos una entrada $\mathbf{x} \in \mathbb{R}^n$ y una salida $y \in \mathbb{R}$ y que queremos una función f que tome a \mathbf{x} como entrada, y de como salida una estimación \hat{y} . La función de costo J calcula el error entre y y \hat{y} .

Si suponemos que este problema se puede tratar como una regresión lineal, tendremos que f deberá ser de la forma expresada en la ecuación (2.1). La forma final del perceptrón incluye una función de activación g usando la ecuación (2.2).

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b \quad (2.1)$$

$$f(\mathbf{x}) = g(\mathbf{w}^\top \mathbf{x} + b) = \hat{y} \quad (2.2)$$

Para minimizar el valor de la función de costo J , los valores de $\mathbf{w} \in \mathbb{R}^n$ y b deberán estar en una configuración óptima.

La configuración óptima dependerá del problema que se requiera solucionar, tres ejemplos simples son los de los operadores AND, NAND y OR, definidos como:

$$\text{AND}(a, b) = \begin{cases} 1, & \text{si } a = b = 1 \\ 0, & \text{sino} \end{cases}$$

$$\text{NAND}(a, b) = \begin{cases} 0, & \text{si } a = b = 1 \\ 1, & \text{sino} \end{cases}$$

$$\text{OR}(a, b) = \begin{cases} 0, & \text{si } a = b = 0 \\ 1, & \text{sino} \end{cases}$$

Donde $a, b \in \{0, 1\}$, para estos operadores existe mas de una configuración que puede representar a cada uno de ellos, como las expuestas en el cuadro 2.1 con la función de activación de la ecuación (2.3).

	\mathbf{w}	b
AND	$[1, 1]^\top$	-1
OR	$[1, 1]^\top$	+1
NAND	$[-1/2, -1/2]^\top$	+1

Cuadro 2.1: Configuraciones de los operadores AND, NAND y OR.

$$g(z) = \begin{cases} 1, & \text{si } z > 0 \\ 0, & \text{sino} \end{cases} \quad (2.3)$$

2.2.1.4. Perceptrón Multicapa

Un solo perceptrón es capaz de representar cualquier función que tenga una dependencia lineal con su entrada; sin embargo, es imposible hacerlo si sucede lo contrario.

Tómese como ejemplo al operador XOR, cuya definición es

$$\text{XOR}(a, b) = \begin{cases} 1, & \text{si } a \neq b \\ 0, & \text{sino} \end{cases}$$

No existe *ninguna* configuración para \mathbf{w} y b que sea capaz de representar a esta función con un solo perceptrón. Por otra parte, es posible representar a esta función con tres perceptrones.

Esto se puede lograr usando las configuraciones que representan a los operadores NAND, OR y AND en conjunto. Para esto consideremos a la matriz $\mathbf{W} = [\mathbf{w}_{\text{NAND}}, \mathbf{w}_{\text{OR}}]^\top$ y el vector $\mathbf{b} = [b_{\text{NAND}}, b_{\text{OR}}]^\top$ ¹.

En la solución se utilizan dos capas de perceptrones, la primer compuesta por los perceptrones que representan a NAND y OR, y la segunda capa por el perceptrón que representa a AND.

$$\mathbf{h}^{[1]} = f^{[1]}(\mathbf{x}) = g(\mathbf{z}) = g(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad (2.4)$$

$$h^{[2]} = f^{[2]}(\mathbf{z}^{[1]}) = g(\mathbf{w}_{\text{AND}}^\top \mathbf{a}^{[1]} + b_{\text{AND}}) \quad (2.5)$$

Esta solución es equivalente a la ecuación (2.6).

$$\text{XOR}(a, b) = \text{AND}(\text{NAND}(a, b), \text{OR}(a, b)) \quad (2.6)$$

Existen diversos problemas que tienen naturaleza no-lineal donde es necesario realizar una transformación de la entrada en diferentes fases, de manera que se pueda resolver en la fase final como un problema de regresión lineal simple, tal como se muestra en las ecuaciones (2.4) y (2.5).

De esta idea surge el concepto de perceptrones multicapa (MLP), que consta de una función de la forma:

$$f^*(\mathbf{x}) = f^{[l]}(f^{[l-1]} \cdots (f^{[1]}(\mathbf{x}))) \quad (2.7)$$

que representa a un MLP de l capas, cada capa puede tener una cantidad diferente de neuronas.

El perceptrón multicapa es la arquitectura más conocida de ANN, esta consiste en apilar múltiples capas de neuronas artificiales (o perceptrones), y se puede representar como la figura 2.1.

2.2.2. Programación Orientada a Objetos

El paradigma de la programación orientada a objetos nace en los años 60 en el campo de la inteligencia artificial, y esta basado en el concepto de marcos (ó *frame*). Un marco es una estructura de conocimiento que describe un objeto específico y contiene múltiples ranuras (*slots*) con especificaciones acerca del objeto.

Este paradigma de programación es mundialmente famoso porque nos permite modelar software de manera similar a la realidad por medio de objetos y relaciones entre ellos. Un objeto es la representación de una entidad física ó conceptual que tiene propiedades, comportamientos y relaciones.

Los objetos se crean a partir de una clase. Las clases son plantillas vacías que representan los objetos en términos de sus atributos (propiedades y relaciones) y métodos (comportamientos).

2.2.2.1. Herencia

La herencia es una relación que tiene la programación orientada que nos permite abstraer los atributos y los métodos que comparten un conjunto de clases. La herencia posibilita crear clases a partir de otras existentes especificando nuevas características y reutilizando otras.

El uso de la herencia nos permite volver nuestro modelo más expresivo, el código reducido y factorizado.

¹La notación $\boldsymbol{\theta}$ es equivalente a $(\mathbf{w}; b)$, de manera que $\boldsymbol{\theta}\mathbf{x} = \theta_0x_0 + \theta_1x_1 + \cdots + \theta_nx_n$, donde $x_0 = 1$ ya que $\theta_0 = b$.

2.2.2.2. UML – Diagrama de clases

UML (*Unified Modeling Language*) es un lenguaje que se usa para modelar un sistema, desde diferentes puntos de vista: aspectos y niveles.

En este caso concreto vamos a utilizar el diagrama de clases. Este diagrama permite mostrar la composición de un clase (i.e. atributos y métodos) y las relaciones que tienen las clases entre sí.

2.2.3. Código Limpio (*Clean Code*)

Cuando hablamos de código limpio nos referimos a las buenas prácticas de programación que tiene un desarrollador al momento de escribir código. Siguiendo estas prácticas se logra que el código sea más legible y mantenible.

Existen varias prácticas que un desarrollador puede seguir para escribir código limpio, en esta sección describiremos las que consideramos más importantes y las que usamos en nuestro *framework*.

1. **Nombres descriptivos:** Cuando se habla de nombres descriptivos se hace referencia a que el nombre describa lo que el código representa. Así no es necesario agregar documentación porque los nombres atributos y la firma de los métodos los documentan adecuadamente. Por ejemplo, si nuestro texto dice que tenemos un numero n de neuronas nosotros normalmente escribimos `int n` pero lo correcto es que escribamos `int numberOfRowsNeurons`, de esa manera la persona que lee el código sabe que significa esa variable.
2. **Métodos de no más de una pantalla:** Lo más recomendable es que cuando estemos escribiendo un método, ocupe máximo una pantalla lo que equivale a unas 15 líneas de código. Cuando excedemos este límite normalmente estamos realizando más de una acción en el método, además de que dificulta su lectura, por lo que se podría partir en métodos más pequeños.
3. **Clases con una sola responsabilidad:** En general lo más recomendable es seguir todos los principios SOLID, pero este (S) es uno de los más importantes, ya que nos indica que una clase tiene que tener un único objetivo. Esto nos facilita ubicar qué parte de la lógica va en cada clase.

Todos estos conceptos fueron tomados de *Clean Code* (Martin, 2008).

2.3. Trabajos Relacionados

Existen muy pocas publicaciones que tienen aproximaciones orientadas a objetos para poder representar modelos de redes neuronales.

En Caicedo Torres (2010) se realiza la implementación de una librería de redes neuronales para facilitar su enseñanza que incluye los conceptos de perceptrón, perceptrón multicapa, neurona, y algunos tipos de redes no supervisadas. Sin embargo el modelo presentado es muy elemental, no se logran apreciar muchos componentes básicos de una red neuronal: la función de costo, la función de activación, métodos de optimización, entre otros. A su vez este modelo tampoco es extensible, no es posible agregar nuevos conceptos debido a que no es clara la responsabilidad de cada clase. Véase la figura 2.2.

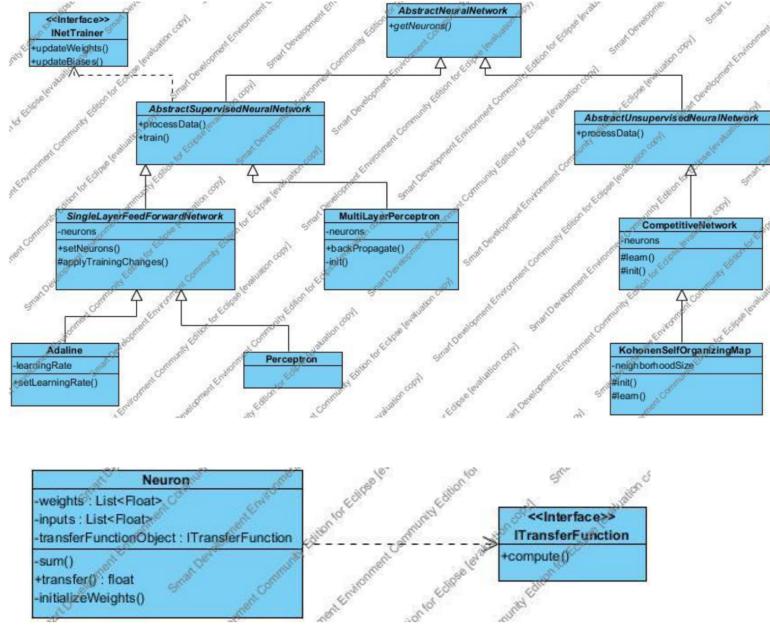


Figura 2.2: Modelo sugerido por Caicedo Torres (2010).

En Valentini y Masulli (2002) se realiza una implementación de redes neuronales orientada a objetos con C++; sin embargo, no usan un lenguaje de modelado estándar por lo que comprender el diseño se dificulta considerablemente. El autor incluye los conceptos básicos de la red, algoritmos de gradiente descendiente, y algunos métodos usados dentro de la red. En este trabajo también observamos que falta ilustrar algunos elementos; como la función de costo, la función de activación, entre otras. Véase la figura 2.3.

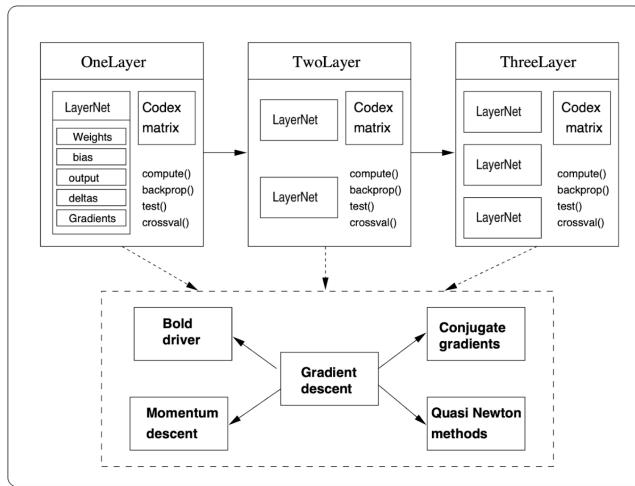


Figura 2.3: Aproximación en Valentini y Masulli (2002).

Ellingsen (1995) es uno de los mejores modelos que encontramos, ya que tiene un buen nivel de claridad y extensibilidad. En este modelo encontramos conceptos como la neuronas y una especie de función de

activación para cada una de estas; también se observan componentes como las capas y la red en si. Usan diagramas similares a UML lo que lo hace fácil de entender el diseño; sin embargo, dada la fecha del trabajo, faltan muchos conceptos que se han ido desarrollando como los métodos de optimización, los métodos de regularización, entre otros. Véase la figura 2.4.

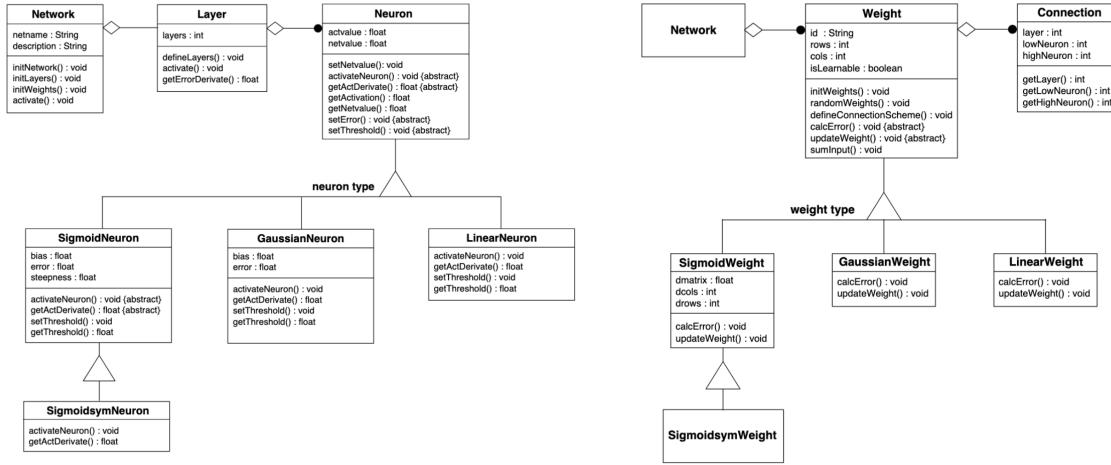


Figura 2.4: Aproximación en Ellingsen (1995).

Aunque Zaitsev (2017) no es un trabajo formal, tiene un modelo que cubre el perceptrón multicapas (o *MLP* por sus siglas en inglés), y funciones de activación que son posibles de extender. Desafortunadamente, no se permite extender las funciones de costo, y al igual que el trabajo anterior Ellingsen (1995) le faltan varios conceptos que se han ido desarrollando últimamente.

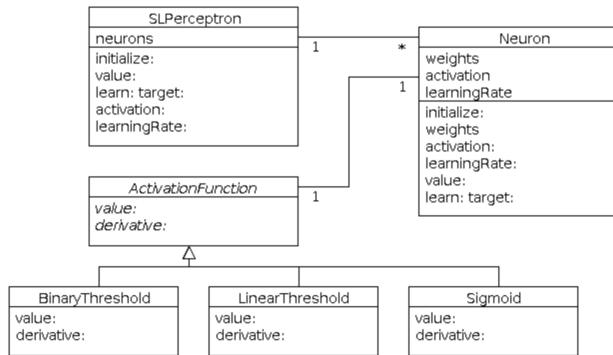


Figura 2.5: Aproximación en Zaitsev (2017).

2.4. Solución

El modelo que proponemos consta de diferentes componentes, cada uno de ellos se enfoca en un tema de *DL* particular. En esta sección explicaremos cada uno de estos temas utilizando como base los componentes de nuestro *framework*.

2.4.1. Diseño

2.4.1.1. Modelos Supervisados

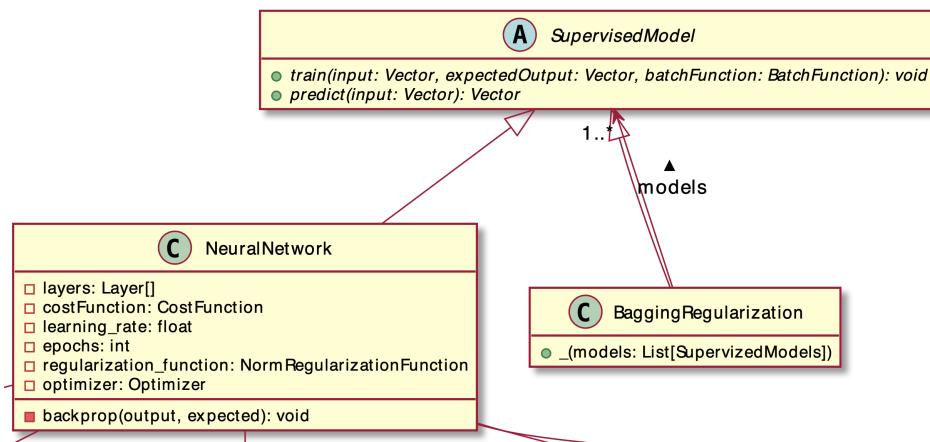


Figura 2.6: Diagrama de clases de la red neuronal con sus propiedades y métodos.

Un modelo supervisado tiene como propósito realizar una aproximación f^* de una función f que depende de una entrada \mathbf{x} tal que $f^*(\mathbf{x}; \boldsymbol{\theta}) \approx f(\mathbf{x})$. Dado que usualmente no se conoce la función f , es necesario ajustar los parámetros $\boldsymbol{\theta}$ del modelo f^* .

Para esto se usan conjuntos de datos \mathcal{D} que contiene una conjunto de tuplas de la forma (\mathbf{x}, \mathbf{y}) donde \mathbf{x} corresponde al conjunto de datos de entrada y \mathbf{y} al conjunto de datos de salidas esperadas.

Existen varios conjuntos de datos para crear un modelo de manera efectiva, el conjunto más grande de datos típicamente es el conjunto de entrenamiento $\mathcal{D}_{\text{train}}$, el cual es usado para aprender los parámetros de los modelos; después tenemos conjunto de validación $\mathcal{D}_{\text{valid}}$, usado para seleccionar entre diferentes modelos; y finalmente el conjunto de pruebas $\mathcal{D}_{\text{test}}$, usado para probar el modelo seleccionado.

En la figura figura 2.6 en la clase de `SupervisedModel` se observan dos métodos, el primer método es `train` que cumple con la función de entrenar el modelo (ajustar los parámetros $\boldsymbol{\theta}$) basado en una matriz \mathbf{X} , una matriz de salida \mathbf{Y} y una función de `batch` la cual se explica más adelante en el apartado 2.4.1.10. El segundo método es `predict`, el cual recibe una matriz de entrada \mathbf{X} y da como retorno una matriz $\hat{\mathbf{Y}}$, que son las estimaciones sobre el conjunto de entrada. Este método se usa en todas las fases: entrenamiento, validación y pruebas.

Cada ejemplo i -ésimo esta contenido en las matriz de entrada $\mathbf{x}^{(i)} = \mathbf{X}_{i,:}$, con su salida asociada a la matriz de salida $\mathbf{y}^{(i)} = \mathbf{Y}_{i,:}$.

Bagging es una técnica para reducir el error de la aproximación por medio de combinar múltiples modelos. La idea es entrenar modelos distintos de manera separada, hacer que cada modelo vote por una salida y decidir cual es la salida final con base en estas salidas. La razón por la que esta técnica funciona es que diferentes modelos no cometerán los mismos errores en el conjunto de pruebas $\mathcal{D}_{\text{test}}$. Las técnicas que emplean esta estrategia se le conocen como métodos de conjunto (ó *ensemble methods* en inglés) (Breiman, 1996).

Generalmente, para k modelos, *bagging* implica crear k conjuntos de datos distintos. La unión de todos los conjuntos de datos tienen el mismo numero de ejemplos que el conjunto original $\mathcal{D}_{\text{train}}$, pero cada conjunto es construido por medio de tomar muestras con reemplazo del conjunto original. Cada modelo i es entrenado sobre el subconjunto i -ésimo.

En la clase `BaggingRegularization` se tienen múltiples modelos que se pueden entrenar de manera secuencial con el método `train` o bien tener los múltiples modelos pre-entrenados. Con el método `predict` se calcula la mejor respuesta, se realiza un promedio de las respuestas de todos los modelos.

Esta solución toma un único conjunto de datos para entrenar todos los modelos. Sin embargo, si se desea entrenar cada modelo con un conjunto de datos distinto, se puede proceder a entrenar cada modelo de manera independiente para luego establecerlos en la clase `BaggingRegularization` en conjunto.

2.4.1.2. Redes Neuronales

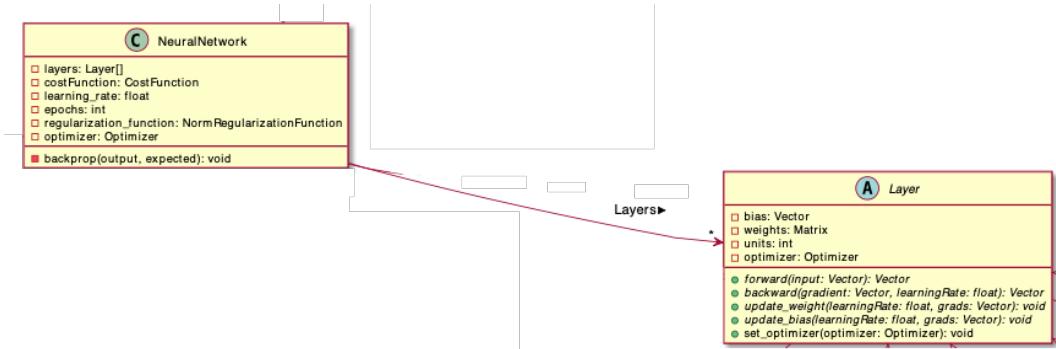


Figura 2.7: Diagrama de las clases `NeuralNetwork` y `Layer`.

Las redes neuronales son un subconjunto de los modelos supervisados y no supervisados; sin embargo, en este trabajo consideraremos los modelos supervisados. Las redes neuronales nacen del trabajo que se realizaron en los años 60 por parte de los neurocientíficos que estudiaban el aprendizaje del cerebro.

Como se puede ver en la figura 2.7 una red neuronal (clase `NeuralNetwork`) tiene una relación con las capas que la conforman (clase `Layer`). Cada capa $i \in \{1, \dots, l = |\text{layers}|\}$ esta conformada por un número de neuronas $n^{[i]}$ (atributo `units`) y sus parámetros: pesos $\mathbf{W}^{n^{[i-1]} \times n^{[i]}}$ (atributo `weights`) y un vector de sesgo $\mathbf{b}^{n^{[i]} \times 1}$ (atributo `bias`).

En general, al hablar de los parámetros del modelo o de una capa específica, se especifica como $\boldsymbol{\theta}$ y $\boldsymbol{\theta}^{(i)}$ respectivamente. En particular, los parámetros mas comunes son \mathbf{W} y \mathbf{b} , de manera que la mayoría de veces se tendrá que $\boldsymbol{\theta} := (\mathbf{W}; \mathbf{b})$.

En la clase `NeuralNetwork` se encuentran los métodos `train` y `predict` por relación de herencia con la

clase `SupervisedModel`. La implementación de `predict` pasa los datos por cada capa, usando el método `forward`, tomando la salida de la capa anterior, o los datos de entrada \mathbf{X} si es la primera capa. Cuando se llega a la capa final (l -ésima capa) del modelo se tendrá como resultado la matriz de estimados $\hat{\mathbf{Y}}^{m \times n^{[l]}}$.

El método `train` efectúa el entrenamiento del modelo, para esto realiza T épocas (atributo `epochs`), para cada una hace una predicción $\hat{\mathbf{Y}}$, calcula el error con la función de coste J (atributo `costFunction`), y propaga la gradiente de la función de coste sobre cada uno de los parámetros $\nabla_{\mathbf{W}^{[i]}} J$ y $\nabla_{\mathbf{b}^{[i]}} J$ de cada capa i -ésima. Cuando se propaga la gradiente se actualizan los parámetros considerando la tasa de aprendizaje, la función de regularización y el optimizador definidos (atributos `learning_rate`, `regularization_function`, `optimizer`). El procedimiento de propagación de la gradiente se realiza con el método `backward`, de manera que en el proceso de entrenamiento al hacer la propagación hacia adelante, se procede a llamar el método y se comienzan a llamar en cadena, en el orden inverso a la propagación hacia adelante.

2.4.1.3. Capas (*Layer*)

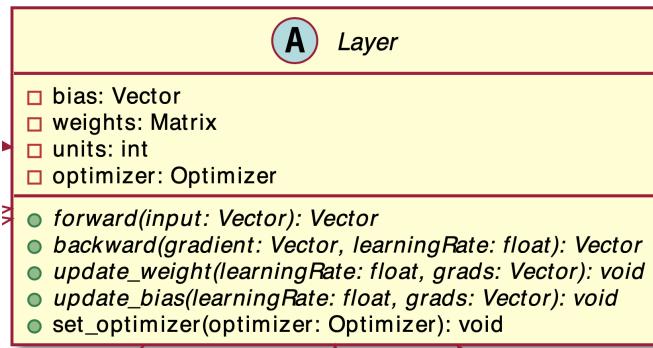


Figura 2.8: Diagrama de clases sobre la clase *Layer*.

Existen diferentes categorías para las capas de una red MLP, en particular se tienen:

1. **Entrada (*Input*):** La capa que reciben la entrada de la red \mathbf{x} , no tienen parámetros asociados, por lo que simbólicamente se considera como $\mathbf{h}^{[0]} = \mathbf{x}$. Por lo tanto, en nuestro *framework* no se toma como una capa. Este se especifica solo como la entrada (parámetro `input` de los métodos de la clase `Layer`).
2. **Ocultas (*Hidden*):** Son todas las capas entre la capa de entrada y la capa de salida (i.e. todas las capas i -ésimas tales que $1 \leq i < l$). Todas contienen parámetros $\theta^{[i]}$, y reciben las salidas de su capa predecesora $\mathbf{h}^{[i-1]}$.
3. **Salida (*Output*):** Esta es la ultima capa del modelo (capa l -ésima), al igual que las capas ocultas tiene parámetros $\theta^{[l]}$. Su salida es la estimación de la red $\hat{\mathbf{y}}$.

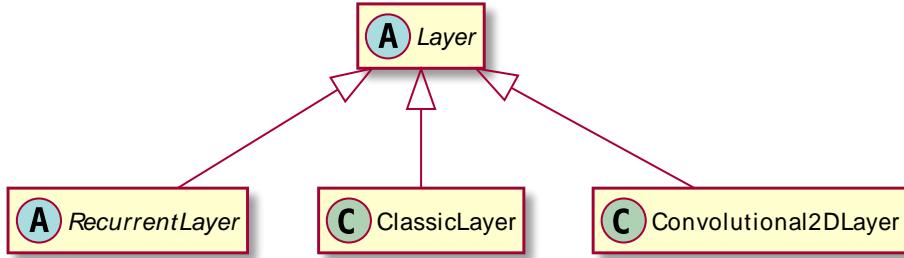


Figura 2.9: Diagrama simplificado del tipo de clases que existen con relación de herencia a la clase `Layer`.

Por otra parte, en este *framework* se implementaron los tres tipos de capas típicas en el aprendizaje profundo, ilustrados en la figura 2.9:

- **Densa (clásica)**: La red mas conocida en aprendizaje profundo (clase `ClassicLayer`).
- **Convolucional**: La red más usada para el procesamiento de imágenes, tiene un conjunto pequeño de parámetros en comparación a las capas densas (clase `Convolutional2DLayer`).
- **Recurrente**: El tipo de red mas usada para el procesamiento de secuencias, al igual que las convolucionales poseen una cantidad pequeña de parámetros en comparación con la cantidad de datos que tiene de entrada (clase `RecurrentLayer`).

2.4.1.4. Propagación hacia adelante (*Forward*)

Este proceso se usa tanto para la fase de entrenamiento como para la fase de predicción.

Para cada capa i -ésima, su salida esta determinada a partir de un vector de entrada que bien puede ser la salida de la capa anterior o ser el vector de entrada de la red, como se expresa en la ecuaciones (2.8) y (2.9).

$$\mathbf{z}^{[i]} = \mathbf{W}^{[i]} \mathbf{h}^{[i-1]} + \mathbf{b}^{[i]} \quad (2.8)$$

Luego se calcula la salida de la capa con la función de activación.

$$\mathbf{h}^{[i]} = g(\mathbf{z}^{[i]}) \quad (2.9)$$

donde $\mathbf{W}^{[i]}$ (atributo `weights`) son los pesos, $\mathbf{b}^{[i]}$ (atributo `bias`) son los sesgos, $\mathbf{z}^{[i]}$ es el *logits* y $\mathbf{h}^{[i]}$ es la salida de la capa.

Todas las capas de la red cuentan con una función de activación g . La función de activación g de la capa de salida tiende a diferir respecto al resto, y depende enteramente del problema que se esta solucionando. (véase la figura 2.10).

2.4.1.5. Funciones de Activación

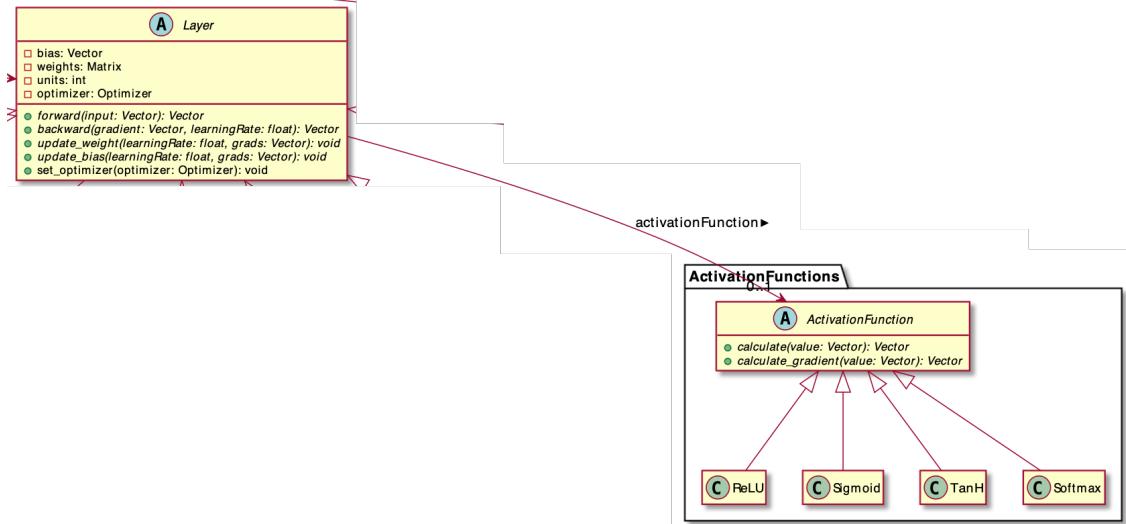


Figura 2.10: Diagrama de la clase `ActivationFunction` de las funciones de activación con sus diferentes implementaciones y su relación con la clase `Layer`.

La función de activación (simbolizada g) es un función no-lineal fundamental dentro de una capa (relación `activationFunction` en la figura 2.10). Esta función decide que tanto se activa una neurona en base a su entrada. Esta función debe ser no-lineal; ya que si es lineal no importa que tantas capas nuestro modelo tenga, este se comportara como un perceptrón de una sola capa ya que al sumarse estas capas darían como resultado solo otra función lineal.

La función de activación en el *framework* tiene dos funciones a implementar:

- `calculate` esta función realiza el calculo de la función de activación sobre la capa
- `calculate_gradient` esta función realiza el calculo de la derivada de la función de activación.

A continuación mostraremos las principales funciones de activación que existen junto con su gradiente. Normalmente se busca que se use la función base como parte del cálculo de su gradiente.

Sigmoid Esta es la función de activación clásica (clase `Sigmoid` en la figura figura 2.10), esto es porque es la más antigua de todas y la que daba mejores resultados. Esta función se define con la siguiente manera:

$$g(z) = \frac{1}{1 + e^{-z}} \quad (2.10)$$

En la ecuación (2.11) se presenta la gradiente de *Sigmoid*.

$$\frac{\partial}{\partial z} g(z) = g(z)(1 - g(z)) \quad (2.11)$$

Tangente hiperbólica Esta función de activación es muy usada en redes recurrentes (clase `TanH` en la figura 2.10) y se define de la siguiente manera:

$$g(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (2.12)$$

En la ecuación (2.13) se presenta la gradiente de \tanh .

$$\frac{\partial}{\partial z} g(z) = 1 - g(z)^2 \quad (2.13)$$

ReLU Esta es la función más nueva de toda la lista (clase `ReLU` en la figura 2.10), y se ha comprobado que es la más efectiva. Esta función se describe como el máximo entre 0 y el valor a calcular, de esta manera se evita que los valores negativos pasen en la red, y se define de la siguiente manera:

$$g(z) = \max(0, z) \quad (2.14)$$

Y su gradiente es:

$$\frac{\partial}{\partial z} g(z) = \begin{cases} 1, & \text{si } z > 0 \\ 0, & \text{sino} \end{cases} \quad (2.15)$$

Softmax Esta función se usa en la capa de salida cuando se requiere hacer multiclasicación, ya que su salida nos entrega un vector de probabilidades, con el cual podríamos calcular cual es la respuesta mas probable, algo muy útil cuando se tiene más de una clase a clasificar (clase `Softmax` en la figura 2.10). Softmax se define con la siguiente ecuación, tomando a $S = \sum_j e^{z_j}$:

$$g(\mathbf{z})_i = \frac{e^{z_i}}{S} \quad (2.16)$$

Y su gradiente se define como:

$$\frac{\partial}{\partial z_j} g(\mathbf{z})_i = \frac{e^{z_j}}{S^2} \begin{cases} (S - e^{z_i}) & \text{si } i = j \\ (-e^{z_i}) & \text{si } i \neq j \end{cases} \quad (2.17)$$

Esta formula al tener como denominador a S , cuya definición es la suma de todos los valores del vector de entrada \mathbf{z} con la expresión e^z , se tiene entonces cada elemento del vector de salida de la función de activación \mathbf{a} que depende de los otros elementos presentes del vector. Al obtener su derivada parcial, cada elemento del vector varia según el resto de elementos presentes.

En esta función tratarse de una gradiente que depende de dos variables (la posición i y j en el vector de *logits* \mathbf{z}), se tiene que la gradiente resultante es una matriz Jacobiana \mathbf{J} a diferencia del resto de funciones anteriormente presentadas.

2.4.1.6. Funciones de Costo

Durante la fase de entrenamiento, una vez realizada la propagación hacia delante, el paso siguiente es calcular el error de la salida $\hat{\mathbf{y}}$ con respecto a los valores esperados \mathbf{y} ; esto se calcula con la función de costo $J(\mathbf{y}, \hat{\mathbf{y}})$ (*Loss Function*).

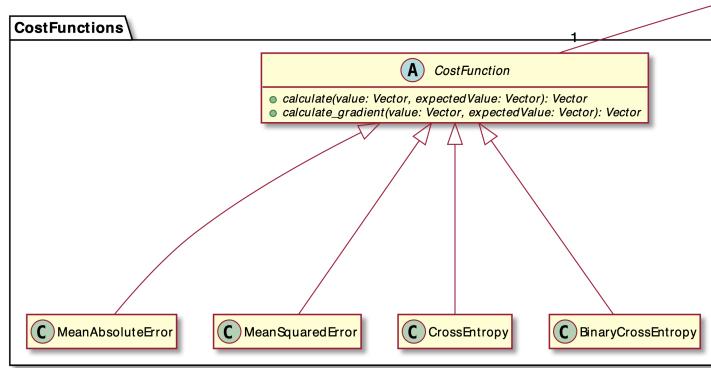


Figura 2.11: Diagrama de clases de las funciones de coste con sus diferentes implementaciones.

La función de costo nos da un estimado de error del modelo y permite calcular una gradiente para optimizarlo. La gradiente la usa el algoritmo de aprendizaje para determinar como ajustar los parámetros. Los diferentes algoritmos de aprendizaje se encuentran en el apartado 2.4.1.10.

En la figura 2.11 se muestra la clase `CostFunction`, esta clase contiene dos métodos referentes a la función de costo J .

- `calculate` esta función realiza el cálculo de la función de costo basada en la salida de la red y en los valores esperados.
- `calculate_gradient` esta función realiza el cálculo de la derivada de la función de costo basada en la salida de la red y en los valores esperados.

Las funciones de costo más comunes se encuentran en el cuadro 2.2.

Formula $J(\mathbf{y}, \hat{\mathbf{y}})$	Nombre	Tipo de problema	Clase
$\ \mathbf{y} - \hat{\mathbf{y}}\ _2^2$	Mean Square Error	Regresión	<code>MeanSquaredError</code>
$\ \mathbf{y} - \hat{\mathbf{y}}\ _1$	Mean Average Error	Regresión	<code>MeanAverageError</code>
$-\mathbf{y} \log(\hat{\mathbf{y}}) + (\mathbf{1} - \mathbf{y}) \log(1 - \hat{\mathbf{y}})$	Binary Cross Entropy	Clasificación binaria	<code>BinaryCrossEntropy</code>
$-\mathbf{y} \log(\hat{\mathbf{y}})$	Multi Cross Entropy	Multiclasificación	<code>CrossEntropy</code>

Cuadro 2.2: Funciones de costo

Las funciones Binary y Multi Cross Entropy se tiene que $\mathbf{y} \in \{0, 1\}^{n^{[l]}}$ y que $\hat{\mathbf{y}} \in [0, 1]^{n^{[l]}}$ al ser f un problema de clasificación de $n^{[l]}$ clases (para Binary es $n^{[l]} = 2$), donde \mathbf{y} es la clasificación verdadera y $\hat{\mathbf{y}}$ es la probabilidad producida por el modelo en un ejemplo en particular. Ya que $\hat{\mathbf{y}}$ son salidas en forma de probabilidad, se debe cumplir $\sum_i \hat{y}_i = 1$.

La elección de cual función de costo J utilizar depende de cada problema: la naturaleza de la función f que intentamos aproximar. En particular para tareas de clasificación, el modelo debe producir probabilidades para obtener la clase con mayor probabilidad, como muestra la ecuación (2.18).

Para tareas de regresión es posible utilizar el resultado del modelo directamente como respuesta o que se realice una transformación ϕ . Un ejemplo es que el valor de respuesta se requiera que sea discreto o que

se realice una transformación de espacios vectoriales para acomodarse al dominio del problema.

$$\hat{y} = \arg \max_{c \in \{1, \dots, C\}} f^*(\mathbf{x})_c \quad (2.18)$$

$$\hat{y} = \phi(f^*(\mathbf{x})) \quad (2.19)$$

Específicamente, la función de costo J trabaja en términos de los resultados \mathbf{Y} de un *batch* \mathbf{X} de m ejemplos (explicados en detalle más adelante), mientras que $J^{(i)}$ trabaja con el ejemplo i -ésimo de \mathbf{X} , en la ecuación (2.20) se ilustra la relación entre ambos.

$$J(\mathbf{Y}, \hat{\mathbf{Y}}) = \frac{1}{m} \sum_{\forall i, \mathbf{y}=\mathbf{Y}_{i,:}; \hat{\mathbf{y}}=\hat{\mathbf{Y}}_{i,:}} J^{(i)}(\mathbf{y}, \hat{\mathbf{y}}) \quad (2.20)$$

2.4.1.7. Propagación hacia atrás (*Backpropagation*)

El método que se usa como parte del proceso de optimización de los parámetros de un modelo es el algoritmo de **backpropagation** (Rumelhart *et al.*, 1986; Goodfellow *et al.*, 2016), que consiste en obtener la gradiente de la función de costo respecto a los parámetros del modelo $\boldsymbol{\theta}$. Primero se obtiene la gradiente de la función de costo respecto a la salida $\nabla_{\hat{\mathbf{y}}} J$, y luego se va iterando hacia atrás por cada capa i -ésima, para así obtener las gradientes $\nabla_{\boldsymbol{\theta}^{[i]}} J$. Para esto se hace uso de la **regla de la cadena** de cálculo.

Definición (Regla de la cadena). *Sean g y f dos funciones diferenciables que tienen dominio y rango en \mathbb{R} . Supóngase que $y = g(x)$ y $z = f(g(x)) = f(y)$, por lo que la derivada $\frac{dz}{dx}$ aplicando esta regla se tiene*

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

El algoritmo luego de obtener las gradientes de la función de costo J respecto a cada parámetro $\boldsymbol{\theta}^{[i]}$ por cada capa i -ésima, realiza un ajuste de estas gradientes según el algoritmo de optimización (véase el apartado 2.4.1.10) que se esté usando, y luego actualiza los valores de los parámetros $\boldsymbol{\theta}^{[i]}$ correspondientes.

En general todos los optimizadores son variaciones respecto a la forma base de gradiente descendiente (Cauchy, 1847), detallado en la ecuación (2.21).

$$\boldsymbol{\theta}_{\{t+1\}}^{[i]} = \boldsymbol{\theta}_{\{t\}}^{[i]} - \alpha \nabla_{\boldsymbol{\theta}_{\{t\}}^{[i]}} J \quad (2.21)$$

donde α es la tasa de aprendizaje y $\boldsymbol{\theta}_{\{t\}}^{[i]}$ son los parámetros de la capa i -ésima en la iteración t .

El *framework* provee el método **backprop** (véase la figura 2.7) para realizar la propagación hacia atrás por toda la red. Los parámetros de esta función son \mathbf{X} las entradas de la red y \mathbf{Y} los valores esperados. El método propaga la gradiente por todas las capas de manera iterativa.

2.4.1.8. Batch

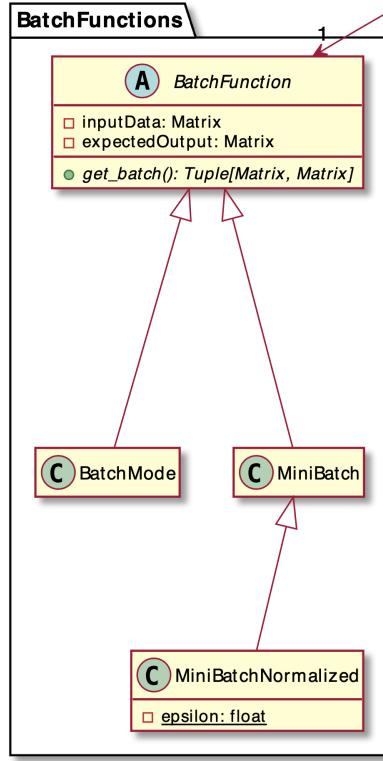


Figura 2.12: Funciones para obtener Batches.

Para entrenar a un modelo con el conjunto $\mathcal{D}_{\text{train}}$ es posible realizarlo de tres maneras: entrenar con un ejemplo a la vez $m = 1$, entrenar con un subconjunto de $m < |\mathcal{D}_{\text{train}}|$ ejemplos, ó entrenar con todos los ejemplos del conjunto de entrenamiento completo $m = |\mathcal{D}_{\text{train}}|$.

Esto significa que al realizar la predicción y propagación de errores (*backpropagation*) durante el entrenamiento, en cada época T (ó *epoch*), se itera por todos los ejemplos de $\mathcal{D}_{\text{train}}$ con una muestra de m ejemplos tomados de manera aleatoria. De esta manera se realizan m predicciones y se calculan m gradientes del modelo completo para luego ser procesados por el algoritmo de aprendizaje.

Cada una de estas muestras de m ejemplos se le conoce como un **batch**. Cada **batch** que se procesa implica una actualización sobre los parámetros del modelo, por lo que se realizan $\lceil |\mathcal{D}_{\text{train}}|/m \rceil$ actualizaciones de parámetros por *epoch*.

Es importante tener en cuenta que los ejemplos del conjunto pueden no ser vectores de una sola dimensión (i.e. $\mathbf{x} \in \mathbb{R}^k$ para cualquier $k \geq 0$); cada ejemplo puede tener más componentes de manera que sea capaz de representar entradas más complejas, i.e. imágenes, imágenes con sus componentes de colores (RGB), secuencias de imágenes con componentes de colores, etcétera. Para generalizar sobre este tipo de entradas, en adelante se especificaran los datos como tensores con notación **X**.

Dado esto, nuestro *framework* utiliza la clase `np.array` de la librería **NumPy**, cuya definición es de tensor, de esta manera, se puede generalizar en el tratamiento de los datos en el *framework*.

En la figura 2.12 se presenta la clase de `BatchFunction`. Esta clase tiene los datos de entrada \mathbf{X} (atributo `inputData`) y los datos esperados \mathbf{Y} (atributo `expectedOutput`) con todos los ejemplos del conjunto de datos de entrenamiento $\mathcal{D}_{\text{train}}$. Tiene la función `get_batch` la cual retorna una tupla que contiene los datos de entrada y los esperados respectivamente.

Para utilizar tamaños de *batch* con $m = 1$ ó $m < |\mathcal{D}_{\text{train}}|$ ejemplos se usa la clase `MiniBatch`, para utilizar el conjunto completo de entrenamiento $m = |\mathcal{D}_{\text{train}}|$ se usa la clase `BatchMode`.

Batch Normalization El algoritmo de Batch Normalization (Ioffe y Szegedy, 2015) es un método de reparametrización adaptativa y esta motivado por las principales dificultades de entrenar modelos muy profundos; entre esas están, las magnitudes no adecuadas de las gradientes, que influyen en cuanto el modelo se acerca ó se aleja de la respuesta correcta. Para mitigar estos problemas el algoritmo opta por que cada *batch* de ejemplos tomados del conjunto de entrenamiento sea normalizado, como se muestra en la ecuación (2.24).

$$\boldsymbol{\mu}_i = \frac{1}{m} \sum_j \mathbf{x}_{j,:} \quad (2.22)$$

$$\boldsymbol{\sigma}_i = \sqrt{\delta + \frac{1}{m} \sum_j (\mathbf{x}_{j,:} - \boldsymbol{\mu}_j)^2} \quad (2.23)$$

$$\mathbf{x}'_{i,:} = \frac{\mathbf{x}_{i,:} - \boldsymbol{\mu}_i}{\boldsymbol{\sigma}_i} \quad (2.24)$$

Donde \mathbf{X} es el *batch* de tamaño m en forma de tensor tomado del conjunto de entrenamiento $\mathcal{D}_{\text{train}}$ y δ es una constante pequeña 10^{-8} que impide que la ecuación (2.24) sea indeterminada evitando que $\boldsymbol{\sigma} = \mathbf{0}$.

En particular, este método es muy deseable al introducir una transformación del espacio de entrada a otro mas estable para el proceso de optimización.

Para utilizar esta técnica sobre los *batch* se usa la clase `MiniBatchNormalized`.

2.4.1.9. Métodos de Regularización

Los conjuntos de entrenamiento $\mathcal{D}_{\text{train}}$ y de prueba $\mathcal{D}_{\text{test}}$ permiten medir la precisión del modelo, e identificar si el modelo esta sobre-ajustado (**overfitting**). El *overfitting* se presenta debido a que el modelo se ajusta únicamente al conjunto pero es incapaz de generalizar sobre el conjunto de datos de prueba $\mathcal{D}_{\text{test}}$.

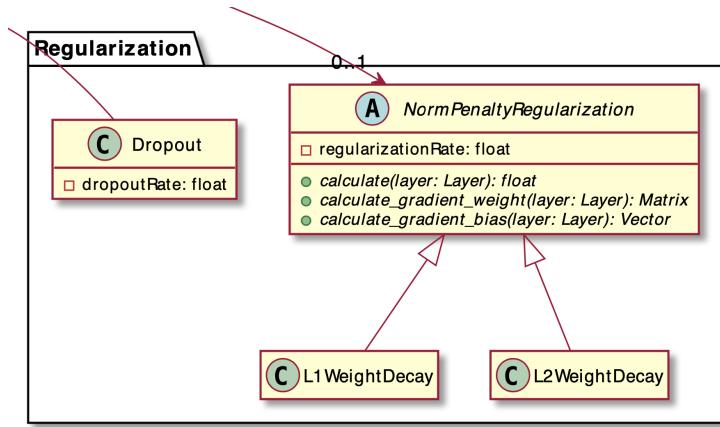


Figura 2.13: Diagrama de clases de los métodos de regularización con sus diferentes implementaciones.

Existen diferentes heurísticas para mitigar el *overfitting*, unas lo hacen por medio de modificar la función de costo J para penalizar a los parámetros θ (clase `NormPenaltyRegularization` en la figura 2.13), otras consisten en hacer modificaciones sobre el modelo introduciendo más capas con el propósito de regular (clase `Dropout` en la figura 2.13), y finalmente, otras consisten en observar las métricas que se hayan establecido y establecer una heurística para detener el entrenamiento.

Regularización sobre los parámetros θ Para regularizar a los parámetros θ se introduce una función adicional $\Omega(\theta)$ sobre la función de costo J (véase la ecuación (2.25)), esto implica que la gradiente de la función de costo respecto a los parámetros $\nabla_{\theta}J$ también cambia. El hiper-parámetro $\lambda \in \mathbb{R}^+$ (atributo `regularizationRate`) controla la cantidad de regularización que se impondrá sobre los parámetros θ del modelo. Las dos funciones mas populares para realizar esto son la regularización L^1 (clase `L1WeightDecay` en la figura 2.13) y L^2 (clase `L2WeightDecay` en la figura 2.13). Estas dos funciones se pueden generalizar con la ecuación (2.26). La ecuación (2.27) es un caso particular.

$$J(\mathbf{y}, \hat{\mathbf{y}}; \theta) = J(\mathbf{y}, \hat{\mathbf{y}}) + \lambda \Omega(\theta) \quad (2.25)$$

$$L^p(\mathbf{x}) = \sqrt[p]{\sum_i x_i^p} \quad (2.26)$$

$$L^\infty(\mathbf{x}) = \max_i |x_i| \quad (2.27)$$

Como se observa en la figura 2.13 para este tipo de regularización se cuenta con tres métodos:

1. `calculate` calcula el valor de penalización de la función de costo con regularización
2. `calculate_gradient_weight` calcula la gradiente de penalización de la función de coste respecto a los pesos de una capa.
3. `calculate_gradient_bias` calcula la gradiente de penalización de la función de coste respecto a los sesgos de una capa.

Las formas de regularización con las ecuaciones (2.26) y (2.27) no están implementados en nuestro *framework*, para implementarlos solo es necesario realizar una extensión de la clase `NormPenaltyRegularization`.

Regularización con Dropout Para regularizar agregando capas al modelo es posible utilizar a Dropout (Srivastava *et al.*, 2014), que consiste en conservar una porción de la salida $\mathbf{h}^{[i-1]}$ de la capa anterior con un a probabilidad $p \in [0, 1]$ (atributo `dropoutRate`) y descartarla con probabilidad $(1 - p)$. Utiliza un vector aleatorio como mascara \mathbf{m} , donde cada uno de sus componentes es tomado de una distribución Bernoulli, $m_j \sim \text{Bernoulli}(p)$, como la ecuación (2.28).

$$\mathbf{h}^{[i]} = \mathbf{m} \odot \mathbf{h}^{[i-1]} \quad (2.28)$$

En el *framework*, la clase `Dropout` en la figura 2.13 se implementó como una capa intermedia que descarta datos basado en una probabilidad `dropoutRate`.

Regularización con métricas del modelo Este método consiste en vigilar las métricas del modelo durante el entrenamiento. Una de las heurísticas más usadas es **early stopping**, que detiene el entrenamiento del modelo una si no ha mejorado su desempeño respecto alguna métrica después de p épocas, a este término se le conoce como **paciencia**. Usualmente la métrica a usar es el error $\epsilon_{\mathcal{D}_{\text{valid}}}$ sobre el conjunto de validación.

Esta forma de regularización actualmente no existe en nuestro *framework*; sin embargo, pueden ser implementadas usando una clase especializada de la clase `callback` definida en el apartado 2.4.1.15. Esta clase al recibir las métricas de desempeño del conjunto de validación ó pruebas puede tomar la decisión de parar el entrenamiento del modelo.

2.4.1.10. Métodos de Optimización

Al entrenar un modelo, dependiendo de la topología de la función de coste existe la posibilidad de que el modelo no converja, o no lo haga de manera eficiente, lo que nos puede llegar a costar tiempo de entrenamiento, o en el peor de los casos que el modelo nunca converja. Por este motivo se introduce el concepto de métodos de optimización.

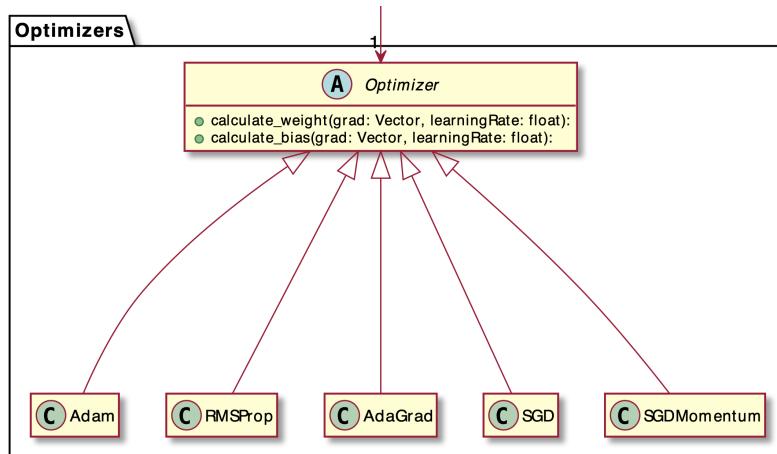


Figura 2.14: Diagrama de clases de los métodos de optimización con sus diferentes implementaciones.

En la figura 2.14 se presenta la clase `Optimizer`, esta clase tiene una relación estricta con la clase

`NeuralNetwork` la implementación mas conocida es SGD. Contiene dos funciones a implementar:

1. `calculate_weight` Esta función calcula que tanto se tiene que actualizar el parámetro de los pesos basado en gradiente (`gradient`) y en la tasa de aprendizaje α (`learningRate`).
2. `calculate_bias` Esta función calcula que tanto se tiene que actualizar el parámetro de los sesgos basado en la gradiente (`gradient`) y en la tasa de aprendizaje α (`learningRate`).

Los métodos de optimización se encargan de actualizar los parámetros θ de las capas en base a sus gradientes, particularmente todos estos métodos utilizan la noción de *batches* al calcularse la gradiente, la gradiente de un *batch* \mathbf{X} de m ejemplos se define como:

$$\mathbf{g}(\mathbf{X}) = \frac{1}{m} \sum_{\forall i, \mathbf{x} \in \mathbf{X}_{i,:}} \nabla_{\theta} J(\mathbf{x}, \mathbf{y}; \theta) \quad (2.29)$$

En adelante, asumase que el conjunto con el que se calcula la gradiente sera el *batch* \mathbf{X} , a menos que se indique explícitamente. \mathbf{g} sera la forma abreviada para la gradiente de la función de costo respecto a los parámetros $\nabla_{\theta} J$.

Gradiente descendiente Consiste en utilizar el tamaño completo del conjunto de entrenamiento $\mathcal{D}_{\text{train}}$ por cada actualización de los parámetros como la ecuación (2.30). Para usar este optimizador se utiliza el optimizador de la clase `SGD` de la figura 2.14, estableciendo el *batch* $\mathbf{X} = \mathcal{D}_{\text{train}}$, de manera que la gradiente calculada será sobre el conjunto entero de entrenamiento. Esto se logra por medio de especificar la clase `BatchMode` de la figura 2.12 dentro de los atributos del modelo.

$$\theta' = \theta - \alpha \mathbf{g}(\mathcal{D}_{\text{train}}) \quad (2.30)$$

Gradiente descendiente estocástica (SGD) Consiste en utilizar un *batch* $\mathbf{X} \subset \mathcal{D}_{\text{train}}$ de tamaño m tomado de manera aleatoria para realizar una actualización de los parámetros como la ecuación (2.31), y donde su tasa de aprendizaje se actualiza típicamente como la ecuación (2.32) y se tiene como restricciones las condiciones de las ecuaciones (2.33) y (2.34). (Clase `SGD` en la figura 2.14).

$$\theta' = \theta - \alpha_k \mathbf{g} \quad (2.31)$$

$$\alpha_k = (1 - \epsilon)\alpha_0 + \epsilon\alpha_{\tau} \quad (2.32)$$

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad (2.33)$$

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty \quad (2.34)$$

con $\epsilon = \frac{k}{\tau}$. Después de la iteración τ es usual dejar a α como una constante.

Gradiente descendiente estocástica con momento Este método se inspira en las leyes de movimiento de Newton simulando tener momento en el recorrido de la gradiente sobre los parámetros. Para esto se utiliza un hiper-parámetro de valor inicial para la velocidad \mathbf{v}_0 y un hiper-parámetro ϵ que regula la velocidad en la que el momento disminuye. (Clase **SGDMomentum** en la figura 2.14).

$$\mathbf{v}' = \epsilon \mathbf{v} - \alpha \mathbf{g} \quad (2.35)$$

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \mathbf{v}' \quad (2.36)$$

AdaGrad Este método adapta las tasas individuales de aprendizaje α de cada parámetros del modelo escalándolos de manera inversamente proporcional a la raíz cuadrada del historial de los cuadrados de la gradiente, este historial es registrado por el vector $\mathbf{r} = \mathbf{0}$ inicializado en ceros (Clase **AdaGrad** en la figura 2.14). Se utiliza la formula de las ecuaciones (2.37) y (2.38) para actualizar los parámetros:

$$\mathbf{r}' = \mathbf{r} + \mathbf{g} \odot \mathbf{g} \quad (2.37)$$

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \frac{\alpha}{\delta + \sqrt{\mathbf{r}'}} \odot \mathbf{g} \quad (2.38)$$

$\delta \approx 10^{-8}$ es una constante pequeña para estabilidad numérica.

RMSProp Este método modifica AdaGrad para mejorar su rendimientos en espacios de parámetros no-convexos por medio de cambiar el acumulado de la gradiente por un ponderado escalado exponencial al introducir un hiper-parámetro ρ (Clase **RMSProp** en la figura 2.14). Las fórmulas corresponden a las ecuaciones (2.39) y (2.40):

$$\mathbf{r}' = \rho \mathbf{r} + (1 - \rho) \mathbf{g} \odot \mathbf{g} \quad (2.39)$$

$$\boldsymbol{\theta}' = \boldsymbol{\theta} + \frac{\alpha}{\delta + \mathbf{r}'} \odot \mathbf{g} \quad (2.40)$$

Adam Es una variante de la combinación de RMSProp y momento, su nombre deriva de “*adaptive moments*” (momentos adaptativos), este algoritmo utiliza dos acumuladores de historia de momento $\mathbf{s} = \mathbf{0}$ y $\mathbf{r} = \mathbf{0}$ con valores iniciales en ceros, tasas de caída de los historiales $\rho_1, \rho_2 \in [0, 1]$, y tiene el registro del numero de iteraciones $t = 0$ que se realizan durante el *epoch*, inicialmente en cero, y se aumentan cada vez que se calcula el valor de las gradientes \mathbf{g} sobre el *batch* \mathbf{X} . (Clase **Adam** en la figura 2.14).

$$\mathbf{s}' = \rho_1 \mathbf{s} + (1 - \rho_1) \mathbf{g} \quad (2.41)$$

$$\mathbf{r}' = \rho_2 \mathbf{r} + (1 - \rho_2) \mathbf{g} \odot \mathbf{g} \quad (2.42)$$

$$\hat{\mathbf{s}} = \frac{\mathbf{s}'}{1 - \rho_1^t} \quad (2.43)$$

$$\hat{\mathbf{r}} = \frac{\mathbf{r}'}{1 - \rho_2^t} \quad (2.44)$$

$$\theta' = \theta - \alpha \frac{\hat{s}}{\sqrt{\hat{r}} + \delta} \quad (2.45)$$

Todos estos métodos de optimización tienen un hiper-parámetro en común, que es la tasa de aprendizaje $\alpha \in [0, 1]$.

2.4.1.11. Funciones de Inicialización

Esta demostrado que inicializar los parámetros de una manera más inteligente puede ayudar a reducir los tiempos de entrenamiento y a mejorar el desempeño del modelo. Adicionalmente, sirve para evitar dos problemas que ocurren frecuentemente: la gradiente desvaneciente, cuando los parámetros son muy pequeños, y la gradiente explotante, cuando los parámetros son muy grandes. por esta razón es bueno utilizar funciones de inicialización.

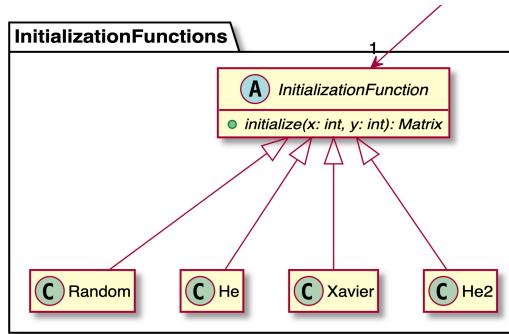


Figura 2.15: Diagrama de clases de las funciones de inicialización con sus diferentes implementaciones.

En la figura 2.15 se presenta la clase `InitializationFunction`, relacionada con la clase `Layer`. Esta clase define el mecanismo de inicialización de cada una de las capas. La función `initialize`, a implementar, tiene como entrada el tamaño de la capa actual y de la capa anterior, y retorna la matriz de pesos junto con los sesgos inicializados. Las capas por defecto se inicializan de forma aleatoria (como se describe más adelante),

Existen muchas técnicas para inicializar los parámetros, en el *framework* están las mas conocidas.

Para la explicación, supóngase que se tienen los parámetros $\theta := (\mathbf{W}^{n^{[i]} \times n^{[i-1]}}; \mathbf{b}^{n^{[i]} \times 1})$ que representan la transformación de la capa anterior $\mathbf{h}^{[i-1]}$ de tamaño $n^{[i-1]}$ a la actual $\mathbf{h}^{[i]}$ de tamaño $n^{[i]}$. Adicionalmente se tiene que \mathbf{R} es una matriz con las mismas dimensiones a θ de variables aleatorias distribuidas uniformemente entre 0 y 1, equivalente a $R_{a,b} \sim U(0, 1)$ para cualquier a y b valido.

Random Es el método mas simple para inicializar los parámetros, esta función nos trae varios problemas, ya que es posible que la red inicie en un sitio muy lejano al mínimo local, lo que puede ocasionar que el modelo sea difícil de optimizar. (Clase `Random` en la figura 2.15)

$$\theta_0 = \mathbf{R} \quad (2.46)$$

He Esta función (He *et al.*, 2015) normaliza los datos aleatorios multiplicándolo por una función que depende del tamaño de la capa. (Clase **He** en la figura 2.15)

$$\theta_0 = \sqrt{\frac{2}{n^{[i]}}} \mathbf{R} \quad (2.47)$$

He (variación) Existe otra variante de He, que toma en cuenta el tamaño de la capa anterior para inicializar los parámetros. (Clase **He2** en la figura 2.15)

$$\theta_0 = \sqrt{\frac{2}{n^{[i]} + n^{[i-1]}}} \mathbf{R} \quad (2.48)$$

Xavier Esta función de inicialización (Glorot y Bengio, 2010) es similar a He solo que divide la función entre $\sqrt{2}$, esta función da excelentes resultados con la función de activación tanh. (Clase **SGDMomentum** en la figura 2.15)

$$\theta_0 = \sqrt{\frac{1}{n^{[i]}}} \mathbf{R} \quad (2.49)$$

Para ver más detalles acerca de las funciones de inicialización recomendamos ver Katanforoosh y Kunin (2019).

2.4.1.12. Redes neuronales densas

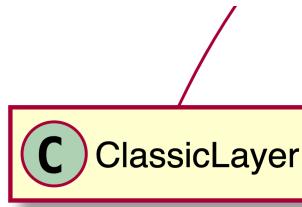


Figura 2.16: Diagrama de clase de redes neuronales densas clásicas **ClassicLayer**.

Esta es la categoría a la que pertenecen las redes que hasta ahora se han presentado. Las redes densas están formadas por capas clásicas, con parámetros dados por una matriz de pesos \mathbf{W} , un sesgo \mathbf{b} y una función de activación g , utilizando la misma aproximación presentada en las ecuaciones (2.8) y (2.9). En nuestro *framework* estas capas están ilustradas en la clase **ClassicLayer**, como se muestra en la figura 2.16, y tienen una relación de herencia con la clase **Layer**, ilustrada en la figura 2.8.

La justificación más recurrente para su uso de múltiples capas clásicas es realizar *aprendizaje de representaciones*. De esta manera no solo es posible aprender mapeos de los valores de entrada a los valores de salida sino también aprender las representaciones en sí (Goodfellow *et al.*, 2016). El objetivo es obtener una representación para la cual es posible generalizar sobre todas las posibles entradas posibles que se pueden tener, esto sin necesidad de observar todos los ejemplos existentes del dominio de entrada.

2.4.1.13. Redes Neuronales Convolucionales

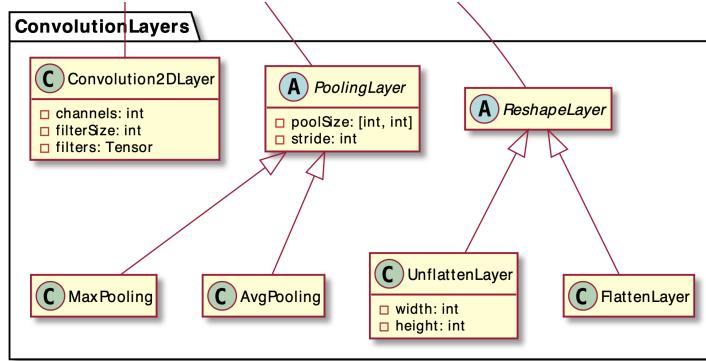


Figura 2.17: Diagrama de clases de las capas del paquete de convolución.

Las redes neuronales convolucionales son un tipo de redes especial para tratar datos multidimensionales, como imágenes y sonidos, estas redes están basadas en la corteza visual del cerebro de los gatos. Una de las más grandes ventajas que tienen estas redes es la expresividad que se genera con la capa de convolución, y la reducción que se obtiene de la capa de *pooling*, lo que nos permite tener muchos menos parámetros y esto reduce el tiempo de entrenamiento.

En particular, las entradas individuales de este tipo de redes generalmente tienen la forma de una matriz \mathbf{X} , diferente para el caso de las redes densas clásicas que como entrada individual tendrían un vector \mathbf{x} . Más importante aún, al tratarse usualmente de procesamiento de imágenes, estas por lo general presentan tres componentes de color (RGB), por lo tanto, la entrada será un tensor \mathbf{X} de al menos tres ejes por cada ejemplo del conjunto original de datos.

Al tener usualmente imágenes como datos de entrada, utilizaremos la convención de imagen \mathbf{I} ó \mathbf{l} en este apartado para representar la entrada de esta redes.

Convolución La capa de convolución se encarga de extraer características de los datos por medio de filtros que se desplazan por los datos realizando una serie de operaciones para sacar diferentes características. Un ejemplo sencillo es cuando de una imagen se extraen los bordes. Es importante recalcar que en este tipo de capas los parámetros son los mismos filtros. (Clase `Convolution2DLayer` en la figura 2.17)

La operación de convolución toma a una imagen de entrada \mathbf{I} y tiene como parámetros a por lo menos un filtro \mathbf{K} de un tamaño usualmente mucho menor a la imagen de entrada. Usualmente al ser \mathbf{I} bidimensional, se utilizará un filtro bidimensional también. Esta operación consta de dar como salida una matriz \mathbf{S} y su fórmula es específica en la ecuación (2.50).

$$\mathbf{S}(i, j) = (\mathbf{I} * \mathbf{K})(i, j) = \sum_a \sum_b I_{a,b} K_{i-a,j-b} \quad (2.50)$$

para cualquier i y j válido.

Ya que las imágenes que se reciben usualmente constan de múltiples canales de entrada, se aplica un filtro distinto por canal. Además, la salida de estas redes tiene más canales que los canales de entrada. Por lo tanto, la cantidad de parámetros es proporcional a la cantidad de canales de entrada C_{in} , las

dimensiones del filtro ($K_{\text{height}}, K_{\text{width}}$) y la cantidad de canales de salida C_{out} . Cada filtro consta de un sesgo. La cantidad total de parámetros se define por la ecuación (2.51).

$$(C_{\text{in}} K_{\text{height}} K_{\text{width}} + 1) C_{\text{out}} \quad (2.51)$$

Una de las más importantes características para el uso de redes convolucionales es el de *compartir parámetros*; lo cual se refiere a usar los mismos parámetros para diferentes zonas del modelo. En las capas densas, cada parámetro solo es usado una vez al calcularse la salida de la capa.

Pooling El proceso de *pooling* se encarga de extraer las características más relevantes de la imagen, de esta manera podemos reducir mucho mas la dimensionalidad de ella (Clase `PoolingLayer` en la figura 2.17). Hay dos tipos de *pooling*: **max pooling** que se encarga de sacar el máximo valor de los datos dentro de la zona de *pooling* (Clase `MaxPooling` en la figura 2.17), y el **avg pooling** que se encarga de sacar un promedio de estos datos (Clase `AvgPooling` en la figura 2.17). El *pooling* más usado es el *max pooling* ya que experimentalmente se ha demostrado que da mejores resultados frente a *avg pooling*.

Cambio de dimensiones

En una arquitectura típica de redes con capas convolucionales se requiere cambiar de dimensiones los datos que fluyen por ella (clase `ReshapeLayer`).

La primera forma de hacer cambio de dimensiones es *aplanar* la matriz \mathbf{X} de entrada a un solo vector \mathbf{x} , esto por cada ejemplar (clase `FlattenLayer`).

La segunda forma es restaurar las dimensiones de la matriz a partir del vector de entrada $\mathbf{x} \in \mathbb{R}^k$ de la capa a una forma definida de matriz \mathbf{X} con dimensiones $(h \times w)$, donde $hw = k$, esto para cada ejemplar (clase `UnflattenLayer`), h corresponde al atributo `height` y w al atributo `width`.

2.4.1.14. Redes Neuronales Recurrentes

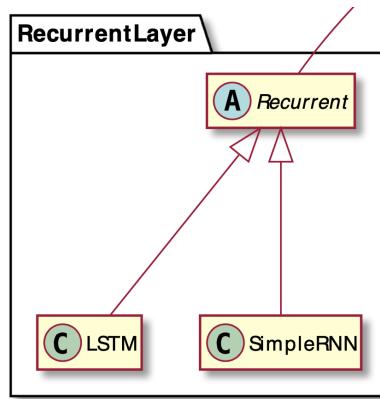


Figura 2.18: Diagrama de clases de las capas del paquete de redes neuronales recurrentes.

Existen datos que tienen una forma de dependencia condicional a otras entradas, una de estas formas puede estar en una secuencia $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(\tau_x)}$ de tamaño variable $\tau_x \in \mathbb{N}$. Por otro lado existen problemas que

no solo cuentan con una secuencia como entrada sino también de una secuencia como salida $\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(\tau_y)}$ de tamaño variable $\tau_y \in \mathbb{N}$.

Las redes *neuronales recurrentes* (RNN) lidian con este tipo de entradas y salidas con estados internos que cambian al procesar una secuencia de entrada, de manera que el estado interno $\mathbf{h}^{[t]}$ en el instante t se define en función de su estado del instante anterior $\mathbf{h}^{[t-1]}$ y su entrada t -ésima de la secuencia, como se muestra en la ecuación (2.52).

$$\mathbf{h}^{[t]} = f(\mathbf{h}^{[t-1]}, \mathbf{x}^{(t)}; \boldsymbol{\theta}) \quad (2.52)$$

En particular al tener diferentes posibles combinaciones para el tamaño de las secuencias de entrada τ_x y salida τ_y , existen entonces cinco variaciones de redes recurrentes unidireccionales².

Uno a uno Para esta variación, el tamaño de las secuencias de entrada como de salida son de un solo elemento, i.e. $\tau_x = \tau_y = 1$. A esta variación se le podría considerar equivalente a una neurona de una red *MLP*.

Uno a muchos En esta variación, a partir de un solo elemento de entrada se genera una secuencia de salidas, i.e. $\tau_x = 1$ y $\tau_y > 1$. Esta variación usualmente es utilizada para la generación de secuencias, un ejemplo es generación de notas musicales a partir de una configuración de entrada.

Muchos a uno Aquí se tiene una secuencia de entradas y un solo elemento de salida, i.e. $\tau_x > 1$ y $\tau_y = 1$. Ejemplos de uso es la clasificación de una secuencia de palabras (una frase) tal que se quiere predecir una cualidad, como clasificación de sentimientos o el estilo de escritura³ (agradecido, confiado, formal, informal, reflexivo, amoroso, triste, acusatorio, preocupado, etcétera).

Muchos a muchos En esta configuración, se tiene una secuencia de entrada y se tiene una secuencia de salida de igual tamaño, i.e. $\tau_x = \tau_y > 1$. Un ejemplo de uso es *Named-entity Recognition*, este consiste en extraer las partes de un texto que se consideran sustantivos, tales como objetos del mundo real, personas, ubicaciones, organizaciones, productos, etcétera.

Muchos a muchos (a largo plazo) Finalmente en esta variación, se tiene una secuencia de entrada y una secuencia de salida de longitud distinta, i.e. $\tau_x > 1$, $\tau_y > 1$ y $\tau_x \neq \tau_y$. Una de los usos que ha tenido esta variación es realizar traducción de maquina que consiste en traducir texto o habla de un lenguaje natural a otro lenguaje natural.

²Esto es que la red solo realiza operaciones en un sentido, desde el instante $t = 1$ hasta el instante $t = \tau$.

³Grammarly es un software de análisis de ortografía en inglés que tiene una función de análisis de estilo para ayudar a quien lo usa a dejar clara su intención.

2.4.1.15. Intervenciones en el entrenamiento y uso de métricas de desempeño

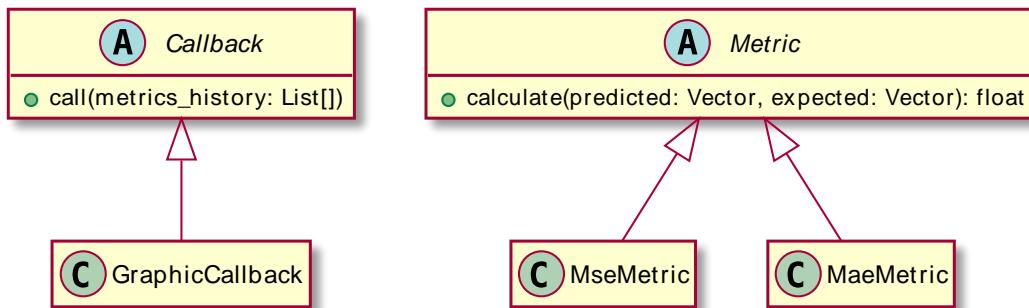


Figura 2.19: Diagrama de las clases **Callback** y **Metric**.

Al entrenar un modelo es importante conocer como este se desempeña a medida que va avanzando en sus iteraciones y en sus épocas, por esta razón debe ser posible intervenir sobre la ejecución del entrenamiento de un modelo por medio de heurísticas basadas en métricas definidas previamente al entrenamiento.

En nuestro *framework* estas intervenciones se pueden realizar por medio de la clase **Callback**, esta contiene el método `call`, el cual recibe el historial de valores de las métricas establecidas en el modelo.

Adicionalmente, para calcular estas métricas se especifica una clase **Metric**s que contiene el método `calculate`, el cual recibe la respuesta predicha por el modelo (parámetro `predicted`) y la respuesta esperada (parámetro `expected`). Las dos métricas implementadas en el *framework* son *MSE* (clase **MseMetric**) y *MAE* (clase **MaeMetric**).

Por cada época se agrega una entrada en el historial de valores de métricas (campo de entrada `metrics_history`) del cual el **Callback** puede tomar una acción.

La clase **GraphicCallback** recibe las métricas de desempeño definidas y procede a hacer una impresión de las métricas en un gráfico, de manera que se pueden visualizar las métricas a medida que van avanzando las épocas de entrenamiento.

2.4.2. Implementación

El modelo descrito en el apartado 2.4.1 lo implementamos en el lenguaje *Python 3.8*. Para simplificar y tener un mejor desempeño en nuestro *framework* utilizamos la librería de *NumPy* (Oliphant, 2006) que es eficiente al realizar cálculos vectoriales debido a que los ejecuta con instrucciones de bajo nivel (en lenguaje *C*) y opera de manera concurrente. Este *framework* al igual que el modelo esta separado en paquetes para que sea sencillo navegar y extender.

Código limpio Aunque *Python* no es un lenguaje tipado⁴, decidimos poner los tipos en la firma de los métodos, de esta manera, el código es mas legible y puede prevenir errores de parte del usuario que usa o extiende el *framework*. El código 2.1 es un ejemplo de código limpio.

⁴Es un lenguaje donde el tipo de valor que guarda sus variables tiene que ser definido antes de asignarle su valor y no se puede violar su especificación al tratar de asignarle un valor de un tipo distinto.

```

1 class ActivationFunction(ABC):
2
3     def operation(self):
4         return np.multiply
5
6     @abstractmethod
7     def calculate(self, value: np.array) -> np.array:
8         pass
9
10    @abstractmethod
11    def calculate_gradient(self, value: np.array) -> np.array:
12        pass
13
14 class Relu(ActivationFunction):
15
16     def calculate(self, value: np.array) -> np.array:
17         return np.vectorize(lambda x: max(0.0, x))(value)
18
19     def calculate_gradient(self, value: np.array) -> np.array:
20         return np.vectorize(lambda x: 0 if x <= 0 else 1)(value)

```

Código 2.1: Ejemplo de código limpio.

Aunque no esta documentado el código, es claro cual es el objetivo de cada método, atributo y clase. Esto es un buen indicativo de código limpio.

2.4.3. Usabilidad

Un ejemplo de arquitectura y código simple con nuestro *framework* está en el código 2.2.

Esta es la implementación de una red neuronal de cinco capas. La red tiene 4 capas ocultas con función de activación *ReLU* y la capa de salida con función de activación *Sigmoid* (líneas 1..7). La función de coste seleccionada es *MSE*, el optimizador es *AdaGrad*, cada *batch* es de tamaño $m = 5$ y se tiene una tasa de aprendizaje $\alpha = 0,025$ (líneas 8..12).

Al crearse la red con `NeuralNetwork` se establece la arquitectura y se puede proceder a entrenar con el método `train` (líneas 16..19).

Como datos de entrada y salida se tienen dos matrices (con la clase `np.array` de la librería NumPy), donde la matriz de entrada es $\mathbf{X}^{k \times 6}$ y la matriz de salida es $\mathbf{Y}^{k \times 2}$, donde k es el numero de ejemplos de todo el conjunto de entrenamiento.

2.5. Evaluación

2.5.1. Plan

Para evaluar que tan práctico y extensible es nuestro *framework*, vamos a tomar tres problemas clásicos, y los vamos a solucionar con redes neuronales construidas con nuestro *framework*. Estas redes neuronales las vamos a ir modificando para obtener mejores desempeños en los problemas, demostrando la maleabilidad de nuestro *framework*.

```

1 layer_1 = ClassicLayer(6, Relu(), Random())
2 layer_2 = ClassicLayer(8, Relu(), Random())
3 layer_3 = ClassicLayer(8, Relu(), Random())
4 layer_4 = ClassicLayer(2, Sigmoid())
5
6 # Especificacion de la arquitectura
7 architecture = [layer_1, layer_2, layer_3, layer_4]
8 cost_function = MeanSquaredError() # Funcion de coste
9 optimizer = AdaGrad() # Optimizador
10 learning_rate = 0.025 # Tasa de aprendizaje
11 iterations = 100 # Cantidad de epochs para entrenar al modelo
12 batch_function = MiniBatch(5) # Especificacion del tamano del batch
13
14 # Definicion de la red neuronal indicando su arquitectura,
15 # optimizador, hiperparametros y variables de entrenamiento
16 neural_net = NeuralNetwork(architecture, cost_function, learning_rate, iterations,
   optimizer)
17
18 # Entrenamiento del modelo
19 neural_net.train(training_data, expected_output, batch_function)

```

Código 2.2: Ejemplo de implementación de una red neuronal en este *framework*.

2.5.2. Caso: *MNIST*

2.5.2.1. Descripción

MNIST (LeCun *et al.*, 2010) es un conjunto de datos que cuenta con mas de 60 mil caracteres de los números del 0 al 9 escritos a mano, cada imagen esta en escala de grises y son imágenes de 28×28 píxeles.

Este es un problema que ha sido usado por miles de artículos para ver que tan buen desempeño tienen sus modelos, por lo que consideramos pertinente evaluar el desempeño de modelos creados con nuestro *framework* frente a este conjunto de datos.

2.5.2.2. Implementación

Para este problema se desarrollaron dos tipos de redes: una enteramente realizada con capas densas, y otra utilizando capas convolucionales.

Para realizar la comparación, se implementaron cuatro variantes: (1) una arquitectura base de 6 capas densas con los diferentes optimizadores disponibles, (2) una arquitectura profunda de 18 capas densas, (3) diferentes funciones de inicialización disponibles sobre la arquitectura base y (4) arquitectura con una capa convolucional, los diferentes tipos de *pooling* y 6 capas densas.

2.5.2.3. Resultados

Nombre	Arquitectura	α	J	Iteraciones	Optimizador	Precisión
Test Optimizadores						
Basic MLP	R50-R20-R14-R24-Si10	0,030	MSE	6	SGD	87,87 %
ADAM MLP	R50-R20-R14-R24-Si10	0,030	MSE	6	ADAM	93,66 %
AdaGrad MLP	R50-R20-R14-R24-Si10	0,030	MSE	6	AdaGrad	85,46 %
RMSprop MLP	R50-R20-R14-R24-Si10	0,030	MSE	6	RMSprop	91,06 %
Test Deep						
ADAM deep MLP	R10-(R10,12)-(R12,4)-Si10	0,030	MSE	5	ADAM	93,28 %
Test Init						
ADAM MLP - Xavier	R10-(R10,3)-(R12,4)-Si10	0,030	MSE	3	ADAM	94,34 %
ADAM MLP - He	R10-(R10,3)-(R12,4)-Si10	0,030	MSE	3	ADAM	91,13 %
ADAM MLP - He2	R10-(R10,3)-(R12,4)-Si10	0,030	MSE	3	ADAM	92,39 %
Test Convolucional						
Conv Avg Pooling ADAM MLP	C1-AP-(R20,5)-Si10	0,030	MSE	4	ADAM	84,26 %
Conv Max Pooling ADAM MLP	C1-MP-(R20,5)-Si10	0,030	MSE	4	ADAM	86,31 %
Conv Avg Pooling ADAM MLP	C1-AP-(R20,5)-Si10	0,045	MSE	4	ADAM	87,37 %
Conv Max Pooling ADAM MLP	C1-MP-(R20,5)-Si10	0,045	MSE	4	ADAM	90,79 %

Cuadro 2.3: Experimentos y los resultados obtenidos durante la fase de testeo del *framework*. La notación resumida que usamos para expresar los resultados se describen a continuación: Notación de capas: **R** = Relu, **T** = TanH, **Si** = Sigmoid, **So** = Softmax, **C** = Convolución, **MP** = Max Pooling, **AP** = Avg Pooling Para no poner la misma capa varias veces introdujimos esta notación: **(R4,3)** = R4-R4-R4, En este problema el α da buenos resultados en el rango [0,03, 0,05].

2.5.3. Caso: Clasificación de textos

2.5.3.1. Descripción

Este conjunto de datos (Zhou, 2019) consiste en ejemplos de frases de texto, cada uno de longitud variable que consiste en clasificar si se trata de una frase positiva o negativa. Contiene 58 ejemplos en el conjunto de entrenamiento $\mathcal{D}_{\text{train}}$ y 20 ejemplos para el conjunto de pruebas $\mathcal{D}_{\text{test}}$.

En particular, se tiene que la frase es una secuencia de n palabras, cada frase simbolizada como $\mathbf{w} = w_1, \dots, w_n$ y tenemos una salida binaria $y \in \{0, 1\}$ simbolizando si es la frase es negativa o positiva.

Frase	Clasificación
i am not at all good	0
i am not at all bad	1
i am good right now	1
i am very bad right now	0

Cuadro 2.4: Ejemplos de clasificación binaria de texto.

2.5.3.2. Diseño

Para adaptar estas entradas de tamaño variable podemos optar por representar cada una de las frases con un esquema de *embeddings* de palabras, que consiste en mapear cada palabra con un diccionario $\mathbf{x}_w = \psi(w)$, donde la dimensionalidad del diccionario va a ser igual a la cantidad de palabras únicas que existen en todo el conjunto de datos, este seria el vocabulario \mathbb{V} .

El problema al tratarse de secuencias se puede intentar resolver con redes recurrentes, para esto es necesario simbolizar a cada frase \mathbf{w} como una matriz \mathbf{X}_w compuesta de cada vector \mathbf{x}_w correspondiente a cada una de las palabras.

2.5.3.3. Implementación

Debido a limitaciones en el alcance, no se realizó la implementación de las redes recurrentes con los que solucionar este problema.

2.5.4. Hallazgos

En nuestro proceso de investigación no encontramos evidencia de un modelo con todas las características como las que exponemos con el nuestro, como mostramos en el apartado 2.3 los modelos propuestos a través de los años solamente incluyen el *MLP* con una capacidad de extensibilidad limitada.

Al buscar información acerca de la función Softmax, encontramos información limitada respecto al cálculo de su gradiente, y en muchas ocasiones hacían la simplificación de la gradiente junto con la función de costo, lo que limitaba la extensibilidad del modelo.

En gran parte de las publicaciones que encontramos, observamos que el alcance de las explicaciones usualmente no incluían conceptos como la definición y el uso de los *batches*, como tampoco se hacia explicación de como se realizaba la actualización de los parámetros con los métodos de optimización al usarlos, usualmente simplificaban su explicación para considerar solo un ejemplo del conjunto de datos.

En ningún artículo durante nuestra investigación se encontró: métodos de optimización, métodos de regularización, redes recurrentes, redes convolucionales. Lo que a nuestro conocimiento hace que seamos los primeros en diseñar un *framework* que une todos estos conceptos para la enseñanza.

Un error conceptual en el *framework* que se encontró posteriormente a la implementación es que el modelo no debería recibir como parámetro la cantidad de épocas por las que se va a realizar el entrenamiento, este debería estar definido únicamente como parámetro en el método de entrenamiento *train*.

Durante la implementación nos encontramos constantemente con problemas en los cálculos multidimensionales. La mayoría de las veces esto era ocasionado por un detalle de implementación de la librería NumPy, llamada *broadcasting*. El *broadcasting* consiste en ajustar automáticamente las dimensiones para permitir cálculos con datos que tienen que ver con dimensionalidad distinta; y, aunque esta característica es útil, también es un gran riesgo ya que el hace el cálculo a pesar de que la dimensionalidad no coincide en todos los ejes haciendo los cambios necesarios sin generar advertencias del cambio. La solución para esto es siempre hacer validación por medio de aserciones en el código de como deberían ser las dimensiones de entrada y salida al realizar cualquier cálculo con la librería y de esta manera mitigar este tipo de errores.

2.6. Conclusiones

Dado que la calidad en materia de soluciones relacionadas a este trabajo dependen enteramente de la capacidad de comprender correctamente los conceptos de estudio por las personas que se adentran a este campo, es importante tener herramientas que permitan facilitar su aprendizaje.

El *framework* desarrollado aquí es un punto inicial para lograr esto. En particular, es importante ver

como todas las relaciones forman el conjunto de técnicas que se aplican en el área de inteligencia artificial, concretamente el aprendizaje profundo.

Durante el desarrollo de esta herramienta se encontraron detalles importantes de la implementación de redes neuronales que usualmente no están dichas explícitamente en la literatura, usualmente se encontraban soluciones informales que solo cuentan detalles pequeños para ilustrar problemas simples o artículos del estado del arte que ya asumen un conocimiento previo, por lo que muchos detalles de implementación son difíciles de inferir.

Aunque este *framework* no es completo, ni es tan eficiente para desarrollar tareas como los son software comerciales, este posee los aspectos que, hasta donde sabemos, no posee ningún otro artefacto en la academia con este nivel de detalle.

Creemos que esta aproximación es un paso en la dirección correcta para el aprendizaje, donde no solo se ilustra sus relaciones sino que también se muestra una implementación funcional de los problemas, donde incluso el uso de buenas prácticas de programación juegan un papel importante para este fin.

2.7. Trabajo futuro

En las redes convolucionales existen más conceptos como el *stride* y *padding* al realizar el cálculo sobre los datos de entrada. Modelar estos conceptos e incluirlos como una extensión dentro del modelo en el componente de redes convolucionales es un trabajo futuro.

Debido a que existen una gran variedad de arquitecturas para redes recurrentes, es deseable introducir mas extensibilidad dentro de este componente al incluir conceptos como:

- Modelos de secuencia a secuencia Codificador–Decodificador
- Omisión de conexiones (*Skip connections*)
- *Leaky Units*
- Composición de unidades recurrentes
- Memoria explícita
- *Echo State Networks*

Finalmente, el entrenamiento de los modelos con *MNIST*, debido a la gran cantidad de ejemplos (60.000), toma mucho tiempo. Como consecuencia, es tedioso realizar experimentos con volúmenes de datos grandes, por ese motivo sería bueno realizar una optimización.

La razón es que en este momento estamos forzando muchas de las fórmulas a que sean vectorizadas, esto es una gran desventaja dado que no estaríamos aprovechando el rendimiento que nos ofrece *NumPy*. Mejorar el rendimiento es también un trabajo futuro.

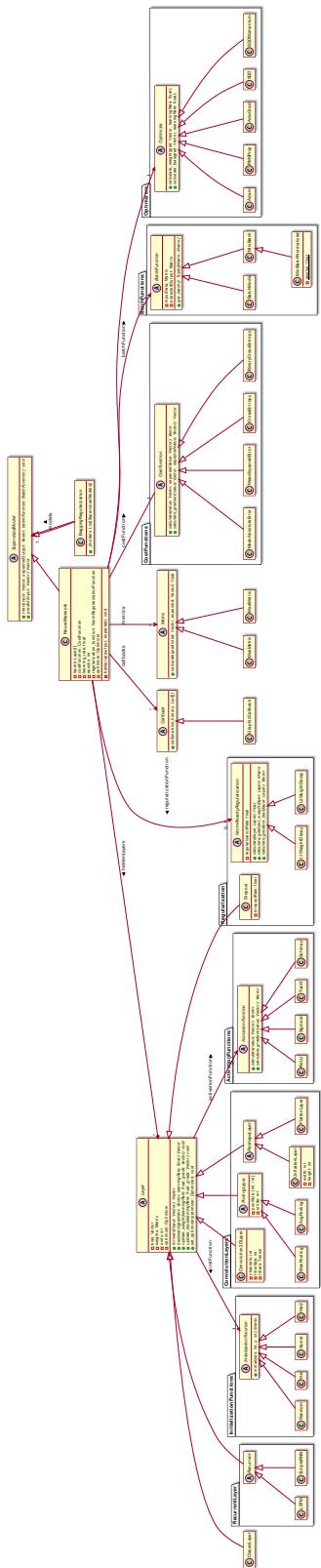


Figura 2.20: Modelo completo.

Capítulo 3

Predicción del caudal Calamar del Río Magdalena con Redes Neuronales

Resumen

El río Magdalena es la fuente hidrológica más importante de Colombia debido a que de él extraemos recursos como agua potable, alimento, energía y también lo usamos como medio de transporte. Incluso en muchas ocasiones el río Magdalena ha provocado catástrofes naturales como sequías e inundaciones afectando a miles de personas. Por estas razones es primordial conocer y predecir el comportamiento del río para tomar decisiones de aprovechamiento y preventivas. En este artículo se presentan dos modelos para estimar el caudal de la estación de Calamar basados en los caudales de cinco estaciones que preceden a esta. El primer modelo es un perceptrón multicapa y el segundo está basado en las redes neuronales secuenciales. Definimos un método para seleccionar la ventana de tiempo de anticipación y el tamaño de la secuencia de predicción más adecuados y realizamos una comparación entre ambos modelos. Finalmente seleccionamos el modelo basado en la arquitectura de perceptrón multicapa debido a que tuvo el mejor rendimiento, este modelo pudo predecir con 15 días de anticipación basado en una secuencia de 30 días, y tiene un RMSE de 867.92 en el conjunto de pruebas y 952 en el de entrenamiento. Estos resultados nos permiten afirmar que las redes neuronales son una buena herramienta para la predicción de caudales.

3.1. Introducción

Los ríos proveen muchos recursos y servicios esenciales para la población humana como el agua potable, los alimentos y la energía. Esta también contribuye al desarrollo y la sostenibilidad de múltiples ecosistemas terrestres y acuáticos.

La cuenca del río Magdalena es el recurso hidrológico más importante de Colombia en términos sociales y económicos. Esta provee de los recursos para obtener agua, alimento, energía y otros recursos al 80 % de la población. La cuenca alcanza una longitud de 1540 kilómetros, proviene desde los glaciares y los bosques de los Andes, y llega hasta su desembocadura en la ciudad de Barranquilla.

El río Magdalena destaca en el continente sur americano al ser el que más transporta sedimentos con una producción media estimada de 700 ton/km^2 al año, esta gran cantidad afecta el 11,8 % del área total de la cuenca. Esto genera una complejidad alta debido a que estos sedimentos reducen la profundidad del río, o incluso llegan a modificar el cauce y la ecología del río, dificultando los estudios que se le realizan.

En general, los ríos nacen en las montañas y fluyen hacia los valles, hacia otros ríos o hacia el mar. A medida que los ríos fluyen, éstos tienen unas zonas naturales de inundación, llamados planos de inundación. El agua de los ríos baja a una velocidad considerable desde su lugar de nacimiento en las montañas. Así, al disminuir la velocidad del agua y dependiendo del relieve de la región, el agua fluirá hacia los lados, inundando las zonas aledañas.

Los desbordamientos pueden tener causas naturales o causas inducidas por actividades humanas. En particular, en las épocas de lluvia o de anomalías climáticas, los ríos reciben grandes cantidades de precipitación que amplían sus caudales, por esto al llegar a los valles, toda esta cantidad de agua no puede ser contenida en los canales de los ríos y, por tanto, se desbordan lateralmente (Escuela de Administración, Finanzas e Instituto Tecnológico, 2020).

Históricamente el río ha presentado múltiples desbordamientos sobre diferentes regiones del país, produciendo inundaciones que afectan a población vulnerable. Por otro lado, cuando se presenta el Fenómeno del Niño se producen sequías, que ocasionan escasez de agua potable y una afectación importante a la industria pesquera.

Consecuentemente, un análisis del historial del caudal¹ podría resultar en la identificación de patrones relevantes que puedan guiar la toma de decisiones de planes de proyectos futuros en el Río Magdalena y de estrategias para disminuir los efectos negativos causados por estas catástrofes naturales.

La mayor parte de las investigaciones que se pueden encontrar en la predicción de cuerpos hidrológicos es la de modelos estadísticos, tales como SMA (*Simple Moving Average*), SES (*Exponential Smoothing*), ARIMA (*Autoregressive Integration Moving Average*), entre otros.

A lo largo del río existen múltiples estaciones hidrológicas que realizan mediciones diarias del caudal del agua en sus diferentes ubicaciones. De estas estaciones la estación de Calamar es una de las más importantes, debido a que en su ubicación la actividad económica como el transporte, la pesca y diversas actividades más dependen del comportamiento del río.

En este artículo presentamos dos modelos de redes neuronales capaces de estimar el caudal de la estación de Calamar, esta predicción se logra al usar los caudales de las estaciones de mayor influencia previas a esta.

3.2. Marco teórico

3.2.1. Hidrología

El Instituto de Hidrología, Meteorología y Estudios Ambientales (IDEAM) de Colombia realiza mediciones regularmente de diferentes variables referentes al río Magdalena. Este ha realizado estas mediciones por más de 50 años, hasta el día de hoy.

El historial de caudales del río Magdalena se construye a partir de observaciones realizadas a diferentes estaciones ubicadas sobre la cuenca, cada una tomada en un intervalo de tiempo fijo. En la siguiente figura se observan seis estaciones del río Magdalena: Purificación, Angostura, Puerto Berrio, Peñoncito, Calamar y Las Flores.

¹Un caudal se define como la cantidad de un fluido que discurre en un determinado lugar por unidad de tiempo.

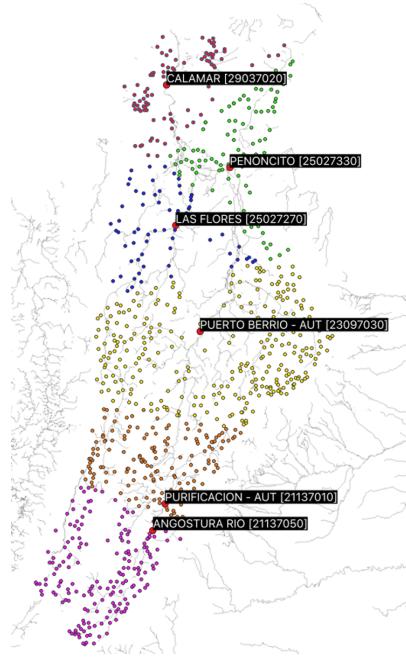


Figura 3.1: Ubicaciones de las estaciones.

El comportamiento de los caudales de la cuenca del río Magdalena son de naturaleza no-lineal (Rojo-Hernández y Carvajal-Serna, 2010). Esto significa que los modelos que sólo incluyen linealidades no pueden usarse para representar las dinámicas de los caudales.

3.2.2. Serie de tiempo

Una *serie de tiempo* (Seymour *et al.*, 1997) es un conjunto de observaciones $x_{\{t\}}$, cada una realizada en un periodo de tiempo específico t . Específicamente, las series de tiempo *discretas* constan de observaciones hechas en intervalos fijos de tiempo.

Para series de tiempo estacionarias², estas pueden llegar a tener la representación del modelo de *descomposición básica*, dado como:

$$x_{\{t\}} = o_{\{t\}} + s_{\{t\}} + z_{\{t\}} \quad (3.1)$$

donde $o_{\{t\}}$ es una función que cambia lentamente y se conoce como el componente de *tendencia*, $s_{\{t\}}$ es una función con un periodo d conocido y se conoce como el componente *estacional*, finalmente $z_{\{t\}}$ es el componente de *ruido aleatorio*.

La figura 3.2 ilustra las tres características de una serie de tiempo estacionaria.

²Una serie de tiempo $x_{\{t\}}$ es estacionaria si su media $\mu_x(t)$ es independiente de t y su covarianza $\text{Cov}_x(t+h, t)$ es independiente de t para cualquier h .

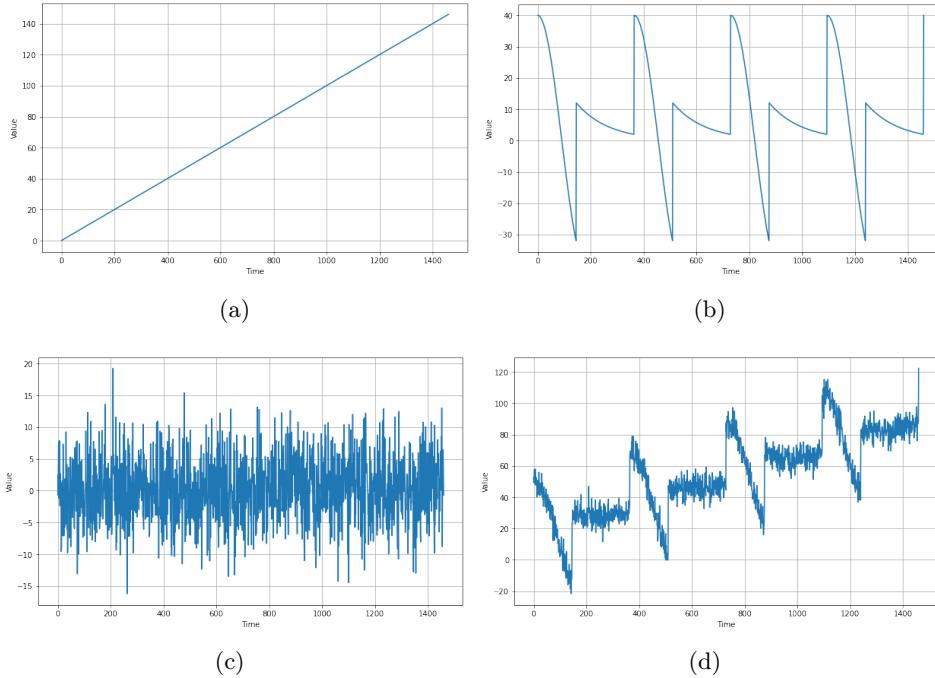


Figura 3.2: Ejemplo de una serie de tiempo. (a) tendencia, (b) temporalidad, (c) ruido aleatorio y la (d) sumatoria de las tres como la serie de tiempo estacionaria discreta.

El historial de los caudales en cada una de las estaciones descritas en el apartado 3.2.1 son series de tiempo. Particularmente estos caudales presentan los diferentes componentes de una serie de tiempo:

- **Tendencia:** se pueden observar tendencias crecientes y decrecientes, dependiendo del año y de otros factores macro-climáticos.
- **Estacionalidad:** se pueden observar estacionalidad según la época del año en la que por el mes el caudal es en promedio mayor o menor.
- **Ruido aleatorio:** según como se tomen los datos del río, sea por instrumentación o por el momento del día en la que se realizan, estos datos tendrán ruido aleatorio de por medio. Las mediciones de caudales se toman en estaciones con ese único fin, por lo que se asume que el tipo de error es aleatorio.

Adicionalmente, el historial de caudales es una serie de tiempo discreta, debido a que la toma de datos es hecha en intervalos de tiempo fijo de aproximadamente un día.

3.2.3. Ventana de promedio deslizante

La *ventana de promedio deslizante* consiste en tomar una serie de tiempo $x_{\{1\}}, \dots, x_{\{t\}}$ y realizar una transformación tal que se genera otra serie de tiempo como:

$$x'_{\{t\}} = \frac{1}{v} \sum_{i=1}^v x_{\{t-i\}}$$

donde $v = \min(t - 1, w)$ y w es el tamaño de la ventana.

Esta heurística tiene como intención reducir el componente de ruido aleatorio z de la serie de tiempo x , de manera que solo queden presentes los componentes $o_{\{t\}}$ y $s_{\{t\}}$. Idealmente se quiere obtener un valor w tal que los componentes $o'_{\{t\}} \approx o_{\{t\}}$, $s'_{\{t\}} \approx s_{\{t\}}$ y $z'_{\{t\}} \approx 0$.

Una serie de tiempo que presenta un componente significativo de ruido aleatorio puede ocasionar problemas para cualquier modelo de estimación, en particular si el modelo es sensible a ligeros cambios sobre la entrada. Por esto, si se reduce su componente de ruido es posible que el modelo tenga un mejor desempeño para estimar entradas futuras.

3.2.4. Modelo ARIMA

El modelo ARIMA (Seymour *et al.*, 1997) (*Autoregressive Integration Moving Average*) se considera como el enfoque más implementado en hidrología y estudios relacionados con variabilidad climática debido a que tiene en cuenta registros de datos estacionarios (con tendencia) (Amaris y Guerrero, 2017).

Este modelo consiste en la combinación de un termino *autoregresivo* (AR) y un termino de *promedio móvil* (MA) con un termino *diferenciador* (I) (Holmes *et al.*, 2020).

Proceso autoregresivo AR(p)

Una serie de tiempo $x_{\{t\}}$ es un proceso autoregresivo AR(p) de orden p si:

$$x_{\{t\}} = \phi_1 x_{\{t-1\}} + \cdots + \phi_p x_{\{t-p\}} + z_{\{t\}} \quad (3.2)$$

donde ϕ_1, \dots, ϕ_p son constantes y $z_{\{t\}} \sim \mathcal{N}(0, \sigma^2)$ (también denominado ruido blanco (WN)).

Proceso de promedio móvil MA(q)

Una serie de tiempo $x_{\{t\}}$ es un proceso de promedio móvil MA(q) de orden q si:

$$x_{\{t\}} = z_{\{t\}} + \theta_1 z_{\{t-1\}} + \cdots + \theta_q z_{\{t-q\}} \quad (3.3)$$

donde $z_{\{t\}} \sim \mathcal{N}(0, \sigma^2)$ y $\theta_1, \dots, \theta_q$ son constantes.

En general, estos modelos se hacen referencia con ARIMA(p, d, q), donde p se refiere al orden del modelo autorregresivo; d al termino de diferenciación; y q al termino de media móvil con q términos de error.

Proceso de diferenciación I(d)

Para el termino del diferencial se debe considerar una evaluación del orden. Los diferenciales pueden ser de primer, segundo, ó hasta orden d , siguiendo la forma de las ecuaciones (3.4) a (3.6) respectivamente.

$$\nabla x_{\{t\}} = x_{\{t\}} - x_{\{t-1\}} = (1 - B)x_{\{t\}} \quad (3.4)$$

donde B se define como el operador de retraso (i.e. $B x_{\{t\}} = x_{\{t-1\}}$).

La diferenciación de segundo orden se define como

$$\nabla^2 x_{\{t\}} = (1 - B)^2 x_{\{t\}} = (1 - B)(1 - B)x_{\{t\}} = (1 - 2B + B^2)x_{\{t\}} = x_{\{t\}} - 2x_{\{t-1\}} + x_{\{t-2\}} \quad (3.5)$$

por tanto, la diferenciación de orden d se define como

$$\nabla^d x_{\{t\}} = (1 - B)^d x_{\{t\}} \quad (3.6)$$

En particular, se puede observar que la diferenciación produce un polinomio de orden d con valores de instantes anteriores.

La estructura general de estos modelos $\{y_{\{t\}} : t \in T\}$, donde T es los instantes de tiempo de la serie, y tiene la forma de la ecuación (3.7).

$$\hat{y}_{\{t\}} = \sum_{i=1}^p \phi_i y_{\{t-i\}} - \sum_{j=1}^q \theta_j z_{\{t-j\}} \quad (3.7)$$

donde ϕ corresponde al coeficiente autorregresivo a determinar, θ el coeficiente de media móvil a determinar, z el término de error y $y_{\{t\}}$ es el registro normalizado de la serie a modelar. En particular, los datos originales $y_{\{t\}}$ se reemplazan por los valores de la diferenciación $\nabla y_{\{t\}}$ mientras se encuentre una tendencia en los datos de la estimación $\hat{y}_{\{t\}}$, esto hasta llegar a una diferenciación de orden d , $\nabla^d y_{\{t\}}$.

La ecuación (3.7) es una mezcla simplificada de las ecuaciones (3.2) y (3.3), con esto mezcla las propiedades los un modelo enteramente autoregresivo con un modelo de promedio móvil, con la mayor parte de sus fortalezas para análisis de series de tiempo y se le introducen componentes de diferenciacion para eliminar los factores.

El objetivo de este modelo es encontrar parámetros para ϕ y θ tales que sea posible modelar el comportamiento futuro por medio de estimar $\hat{y}_{\{t\}}$ a partir de los valores anteriores $y_{\{t-1\}}, \dots, y_{\{t-p\}}$ y $z_{\{t-1\}}, \dots, z_{\{t-q\}}$.

Este modelo solo se considera valido si se cumple la siguiente relación

$$\phi(B)(1 - B)^d x_{\{t\}} = \theta(B)z_{\{t\}}$$

donde ϕ es un polinomio de grado p como la ecuación (3.2) y θ un polinomio de grado q como la ecuación (3.3).

3.2.5. Modelo TRAMO

El modelo TRAMO (Maravall y Peña, 1996) (*Time series Regression with ARIMA noise, Missing observations and Outlier*) es una técnica de interpolación de datos faltantes que son resultado de procesos del modelo ARIMA.

El enfoque principal de TRAMO es combinar por un lado, las facilidades de la detección y corrección automática de valores atípicos y por el otro, la identificación de un modelo ARIMA de una forma eficiente (Tarapuez Roa y Eslava Avendaño, 2011).

Con $\phi(B)$ como polinomio de retraso autorregresivo de la variable $\nabla x_{\{t\}}$ como polinomio de retraso en diferencias de $x_{\{t\}}$, $\theta(B)z_{\{t\}}$ representa un polinomio de retraso de promedios móviles en $z_{\{t\}} \sim \mathcal{N}(0, \sigma^2)$.

TRAMO asume que cualquier polinomio de retraso puede incluir operadores estacionales según la periodicidad y longitud de los datos trabajados, en consecuencia, cuando TRAMO trabaja de forma automática frecuentemente propone modelos SARIMA, la especificación de estos polinomios es señalada a continuación.

$$\phi(B) = (1 - B)^d(1 - B^s) \quad (3.8)$$

A diferencia del modelo ARIMA, SARIMA (*Seasonal ARIMA*), toma un argumento adicional s que establece el componente estacional del modelo. Para TRAMO, s esta definido como el numero de observaciones al año (Seymour *et al.*, 1997; Tarapuez Roa y Eslava Avendaño, 2011).

3.2.6. Red neuronal artificial

Una *red neuronal artificial* (ANN por sus siglas en inglés) es un modelo paramétrico de Aprendizaje Automático (ML por sus siglas en inglés) usada para aproximar la distribución de un conjunto de datos. Reciben este nombre ya que se inspiran en la estructura de las redes neuronales del cerebro. Las redes neuronales logran esto al realizar una estimación inicial y realizar un ajuste de sus parámetros basado en una métrica de desempeño.

La arquitectura más conocida de ANN es la del perceptrón multicapa (MLP por sus siglas en inglés), esta consiste en apilar múltiples capas de neuronas artificiales (o perceptrones), y se puede representar como la figura 3.3.

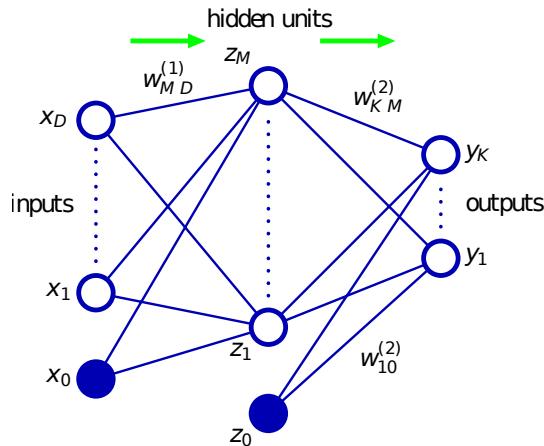


Figura 3.3: Red perceptrón multicapa (MLP). Tomado de Bishop (2006).

Redes Neuronales Recurrentes Las *redes neuronales recurrentes* (RNN por sus siglas en inglés) son redes neuronales especializadas que toman una secuencia de datos como entrada, como las series de tiempo. Dependiendo de la longitud que tenga la secuencia de entrada, el modelo puede experimentar problemas para representar dependencias de largo plazo. Los modelos que han demostrado mitigar este problema de manera más eficaz son las redes LSTM (*Long-Short Term Memory*) y GRU (*Gated Recurrent Unit*).

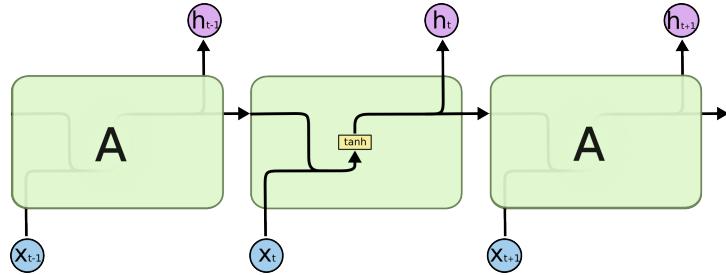


Figura 3.4: Red neuronal recurrente (RNN). Tomado de Christopher (2015).

Long-Short Term Memory (LSTM)

Las redes LSTM (Hochreiter y Schmidhuber, 1997) son un tipo de RNN más especializada. Estas a diferencia de las redes RNN convencionales añaden una entrada *self-loop* conocida como el *cell state* (como se muestra en la figura 3.6), y su propósito es permitir el flujo de información del pasado, donde su entrada es controlada para olvidar o recordar características de las entradas anteriores en la secuencia, aunque estos dos aspectos son controlados de manera independiente dentro de la red. Este mecanismo disminuye considerablemente la dificultad para la red de representar dependencias de largo plazo, mientras que también se mantienen las dependencias de corto plazo. La representación de este tipo de redes se puede ver en la figura 3.5.

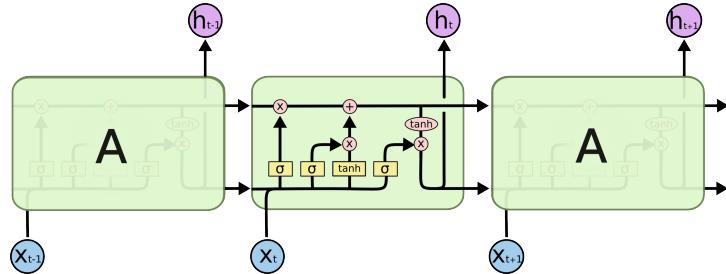


Figura 3.5: Red Long-Short Term Memory (LSTM). Tomado de Christopher (2015).

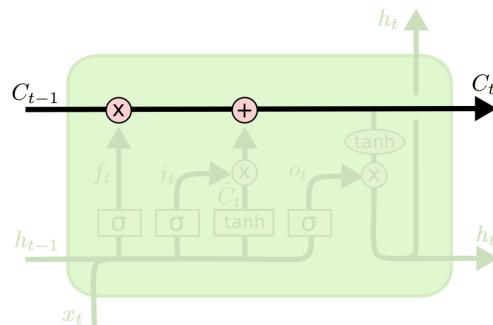


Figura 3.6: Flujo del *cell state* en una red LSTM. Tomado de Christopher (2015).

Gated Recurrent Unit (GRU)

Las redes GRU (Chung *et al.*, 2014) son otra variante de redes LSTM que mitigan el problema de las

dependencias a largo plazo. Su principal diferencia con las redes LSTM es que se usa un solo mecanismo en la entrada del *cell state* que controla el factor para olvidar o recordar características de entradas anteriores. La representación de esta red se puede ver en la figura 3.7.

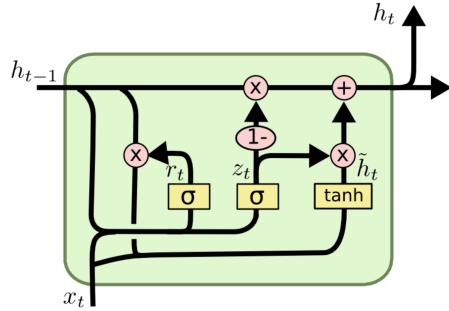


Figura 3.7: Red Gated Recurrent Unit (GRU). Tomado de Christopher (2015).

3.3. Trabajos relacionados

En los últimos años ha crecido la investigación relacionada con la estimación de los caudales de los ríos por medio de diferentes métodos estadísticos y de aprendizaje automático.

3.3.1. Métodos estadísticos

En Aguilar Villena (2016) se intenta predecir el caudal del río Chira en Perú aplicando el modelo ARIMA con el objetivo de identificar el periodo óptimo para realizar descolmatación³ de dicho embalse. Esta aproximación cuenta con un historial del caudal bastante amplio (65 años).

En Rojo-Hernández y Carvajal-Serna (2010) se utiliza un modelo periódico de predicción basado en el Análisis Espectral Singular (AES) usando ventanas de predicción de tres y seis meses aplicado sobre los caudales de los ríos San Carlos, Río Grande II, Guatapé, Magdalena, Guavio y Batá. Los datos que se usan para realizar las predicciones de los caudales son las series de los caudales junto con variables macro-climáticas como algunos de los índices macro-climáticos obtenidos del National Climatic Data Center. Sus resultados parecen indicar que sus modelos reproducen de manera aceptable las principales características estadísticas de la serie de caudales.

En Amaris y Guerrero (2017) realizan una predicción de los volúmenes anuales del río ($m^3/año$) para la estación de Calamar. Para esta tarea se realiza una estimación de un modelo ARIMA para el análisis de series de tiempo de volúmenes anuales en el Río Magdalena con registros de la estación de Calamar. Sus resultados muestran que aunque este modelo no es capaz de simular el comportamiento exacto en el tiempo, son una buena herramienta para realizar una aproximación.

³La descolmatación consiste en retirar la sedimentación, o el material sólido que transporta las corrientes de agua hacia los ríos. Limpieza de canales.

3.3.2. Métodos con redes neuronales

En Veintimilla-Reyes y Cisneros (2015) nos encontramos con un modelo de redes neuronales que intenta predecir el caudal a partir de datos meteorológicos como la precipitación. Por desgracia en este modelo no hay evidencia que la red tenga una buena generalización al no incluir un conjunto de pruebas y tener un numero de iteraciones tan elevado para una red así de simple y un conjunto de datos tan pequeño. Por lo que suponemos que el modelo presenta sobre-ajuste. Adicionalmente los datos que presentan no son claros.

En Espinoza *et al.* (2018) se realiza la comparación de un modelo estadísticos junto con una arquitectura simple de red neuronal, sin embargo la arquitectura planteada contiene muy pocos parámetros debido a que la red presentada solo contiene solo 3 valores de entrada, 2 dos valores en una capa oculta y una sola salida. Tampoco especifican las entradas y las salidas de la red, por lo que no es posible saber que datos de entrenamiento usaron. A su vez no se evidencian métricas que confirmen el desempeño descrito.

En Tanty y Desmukh (2015) muestran distintos casos de uso en los que las redes neuronales pueden servir para resolver problemas de hidrología como escorrentía de drenaje, predicción de caudales, calidad del agua y modelos de agua subterránea. Con respecto a la tarea de estimación del caudal recomienda tomar como entrada los mismos caudales, y usualmente excluyendo variables como la precipitación.

En conclusión, los artículos que aplicaron redes neuronales desafortunadamente no tuvieron en cuenta los problemas que estas presentan, como el sobre entrenamiento, entre otros. No siguieron la metodología de dividir el conjunto de datos en al menos un conjunto de entrenamiento y un conjunto de prueba. Ninguno de ellos definió una métrica clara para determinar que tan confiable es su modelo, por lo que no existe una forma de comparar resultados y determinar la mejor aproximación. Por otra parte, los resultados de los trabajos basados en el modelo ARIMA y AES demuestran robustez en sus resultados.

3.4. Solución

3.4.1. Especificación del Problema

La estación Calamar es una de las estaciones más importantes por su ubicación geográfica, por la actividad socio-económica que existe en el sector y por todos los estudios que se le han realizado.

Por este motivo el problema que vamos a resolver en este trabajo, es la predicción del caudal de la estación de Calamar basado en las cinco estaciones anteriores, maximizando la ventana de predicción, y reduciendo la longitud de la secuencia necesaria para predecir, sin sacrificar el rendimiento del modelo ya que buscamos que el error promedio del caudal sea menor a $2000\text{ m}^3/\text{s}$.

Formalmente, dado un vector de serie de tiempo $\mathbf{y}_{\{t\}}$, se quiere estimar la serie i -ésima de tiempo después de P instantes para cualquier instante de tiempo t valida $i = t + P$ a partir de un intervalo de tiempo de T instantes previos, como se muestra en la ecuación (3.9).

$$\hat{y}_{c,\{t+P\}} \approx f^*(\mathbf{y}_{*-c,\{t-T\}}, \dots, \mathbf{y}_{*-c,\{t\}}) \quad (3.9)$$

La especificación del vector de serie de tiempo $\mathbf{y}_{*-c,\{t\}}$ son los valores de los caudales de las estaciones del río sin incluir Calamar en el tiempo t , y la estimación $\hat{y}_{c,\{t+P\}}$ es la serie de tiempo de la estación de Calamar desplazada P instantes de tiempo en el futuro.

3.4.2. Descripción de datos

Los datos que tuvimos en cuenta al momento de realizar el modelo fueron los de caudales y precipitaciones porque vimos en los trabajo relacionados que eran los datos más relevantes.

3.4.2.1. Caudales diarios 1990 - 2013

Este conjunto de datos contiene los caudales de las seis estaciones presentes en la cuenca del río: Purificación, Angostura, Puerto Berrio, Peñoncito, Calamar y Las Flores.

Porcentaje de valores no-nulos

Con una métrica para calcular el porcentaje de valores no-nulos respecto a todos los datos del conjunto, se encontró la distribución del cuadro 3.1. Adicionalmente en la figura 3.8 se observa la distribución de vacíos que tiene cada una de las estaciones a través del tiempo.

Estación	Porcentaje
Purificación	99.9870 %
Calamar	99.7653 %
Puerto Berrio	98.7223 %
Angostura	95.7757 %
Peñoncito	91.1343 %
Las Flores	83.0769 %

Cuadro 3.1: Porcentaje de valores no-nulos respecto al total de los datos.

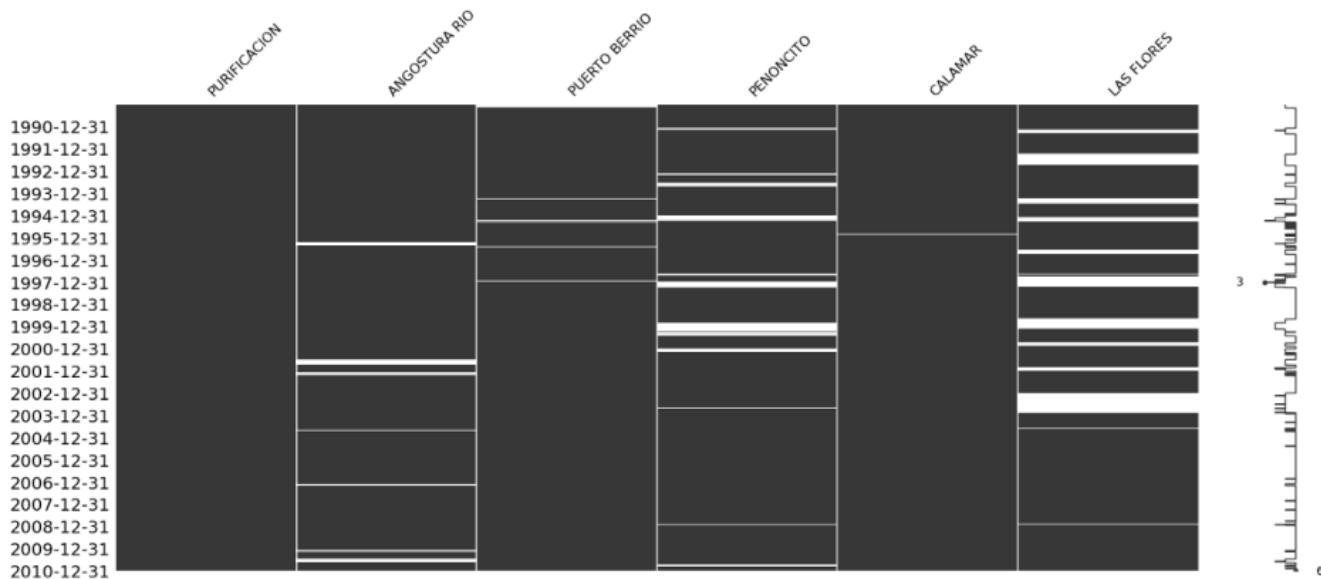


Figura 3.8: Distribución de espacios vacíos dentro del conjunto de datos sobre cada estación de caudal.

Se reviso el nivel de calidad y volumen de los datos faltantes de cada una de las reservas. Entre lo que se puede evidenciar de la figura 3.8, puede verse que los datos faltantes de la estación de Las Flores no son continuos. Por otra parte se tiene el caso contrario para las estaciones de Purificación y Calamar, que

muestran poca interrupción en sus valores consecutivos a través del tiempo.

sin embargo dada la importancia de ubicación, correlación e impacto económico, se escogió a la estación de Calamar como estación a predecir.

Correlación de caudales sobre cada estación

Se realizó una matriz de correlación expuesta en la figura 3.9 entre cada una de las estaciones de la cuenca.

Dentro de las correlaciones obtenidas y de la figura 3.1, se puede observar que entre más cercana sea una estación de otra geográficamente más correlación tienen en sus mediciones.

ID	PURIFICACION	ANGOSTURA	PUERTO BERRIO PEÑONCITO	CALAMAR	LAS FLORES
PURIFICACION	1	0,831244	0,606782	0,361991	0,318153
ANGOSTURA	0,831244	1	0,479747	0,314755	0,253829
PUERTO BERRIO	0,606782	0,479747	1	0,620622	0,444996
PEÑONCITO	0,361991	0,314755	0,620622	1	0,812851
CALAMAR	0,318153	0,253829	0,444996	0,812851	1
LAS FLORES	0,3625	0,318255	0,640826	0,77616	0,692938

Figura 3.9: Matriz de correlación entre caudales de las estaciones.

Análisis del caudal en la estación de Calamar

En la estación de Calamar se cuentan con datos diarios desde el 1ro de enero de 1990 hasta el 31 de diciembre de 2010, como se puede visualizar en las figuras 3.10 y 3.11. En figura 3.10 se ilustra el detalle de los caudales en ese período y en la figura 3.11 se encuentra un gráfico por año de los caudales junto con su media diaria (línea negra), ahí se puede apreciar el componente de estacionalidad general de la serie de tiempo de los caudales.

Entre el año 1990 y 1997 el caudal se mantuvo constante en promedio. En el período de 1997–1998 se puede observar un comportamiento atípico de valores bajos. Este comportamiento parece replicarse en el año 2009. Desde el 2002 en adelante el caudal parece presentar un aumento en tendencia año tras año, exceptuando al 2009. Adicionalmente, también se puede apreciar en el primer y tercer trimestre el caudal disminuye cada año y coincide con el principio del período de sequías, mientras que en el inicio del segundo y cuarto trimestre el caudal aumenta y coincide con los períodos de lluvia.

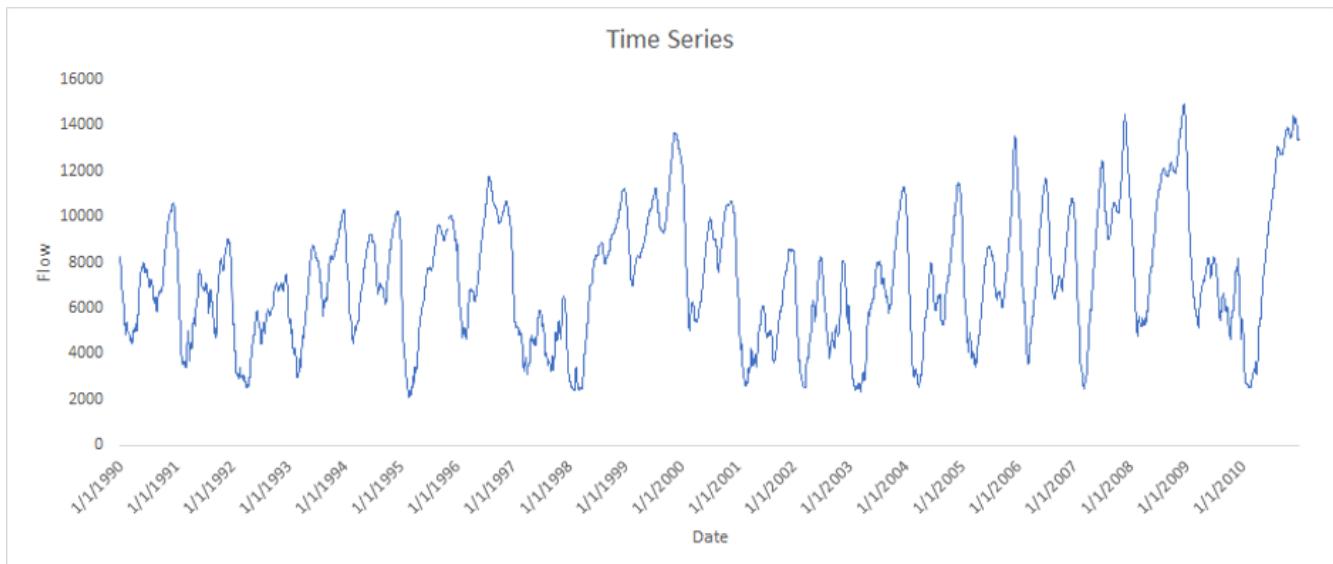


Figura 3.10: Caudales de la estación de Calamar.

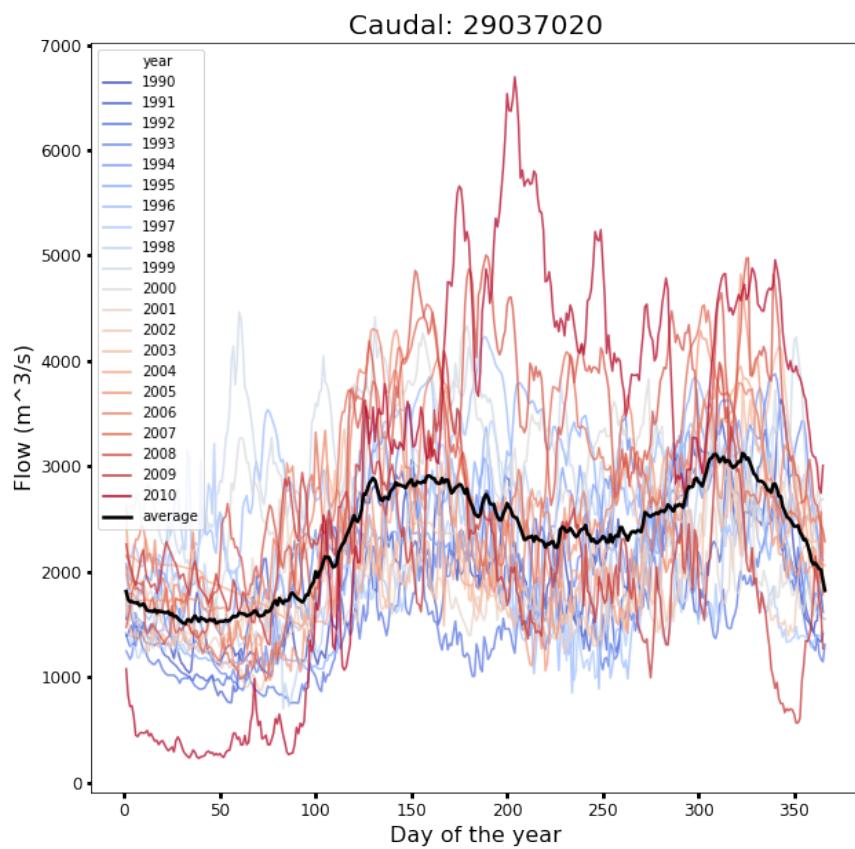


Figura 3.11: Caudales de la estación de Calamar, eje adyacente con un solo año.

3.4.2.2. Pre-procesamiento de datos

Para completar los datos faltantes de los caudales aplicamos el modelo estadístico de TRAMO utilizando el software Gretl (Baiocchi y Distaso, 2003), este software tiene un límite de procesamiento de 600 datos por lo que se realizó el relleno por partes.

Como la componente de ruido aleatorio de las series de tiempo de algunos caudales aparentemente es muy alta, lo que hicimos fue aplicar una ventana de promedio deslizante sobre cada uno de los caudales.

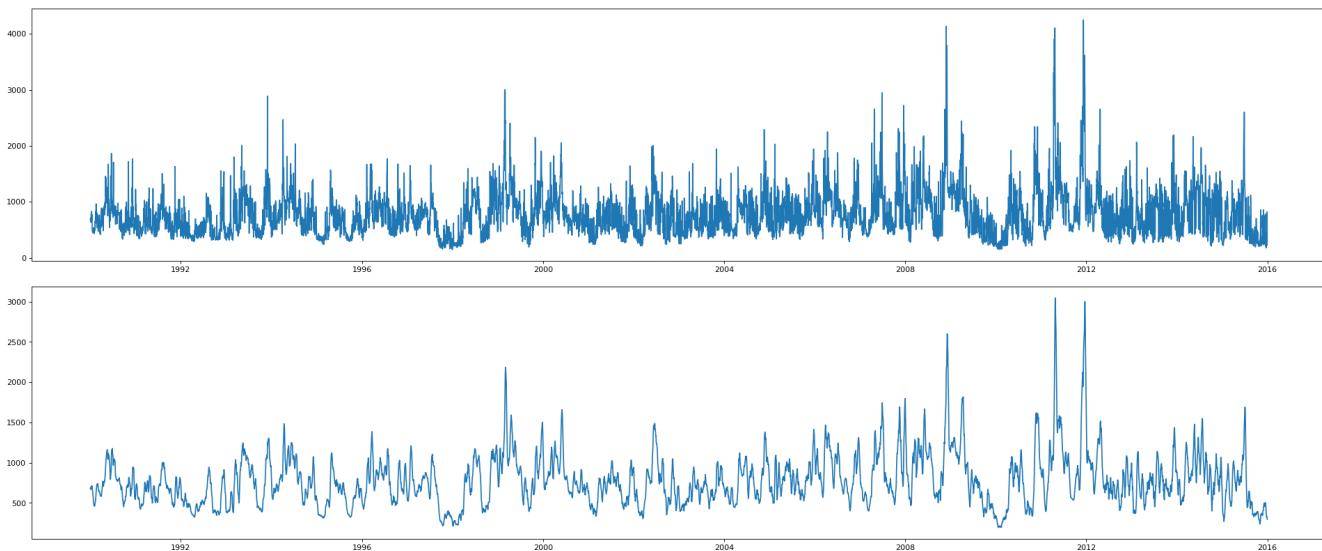


Figura 3.12: Serie de tiempo de una estación con ruido, y después de aplicar el pre-procesamiento de datos con una ventana de promedio deslizante de 15 días ($w = 15$).

3.4.2.3. Precipitación diaria 1990 - 2015

Este conjunto de datos consta de 878 estaciones a lo largo del Río Magdalena desde el 1ro de enero de 1990 hasta el 31 de diciembre de 2015.

Porcentaje de valores no-nulos Lamentablemente este conjunto de datos cuenta con una gran cantidad de datos vacíos como se observa en la figura 3.13 en esta gráfica podemos observar que hay estaciones que tienen hasta un 25 % de datos faltantes. Además de estos datos faltantes, hay un gran porcentaje de datos con el valor igual a 0, creemos que esto se debe a que la recolección de estos datos se realiza en la mayor parte de las estaciones de manera manual y esto es susceptible a errores humanos, constituyéndolo de ser un error sistemático en la toma de las muestras.

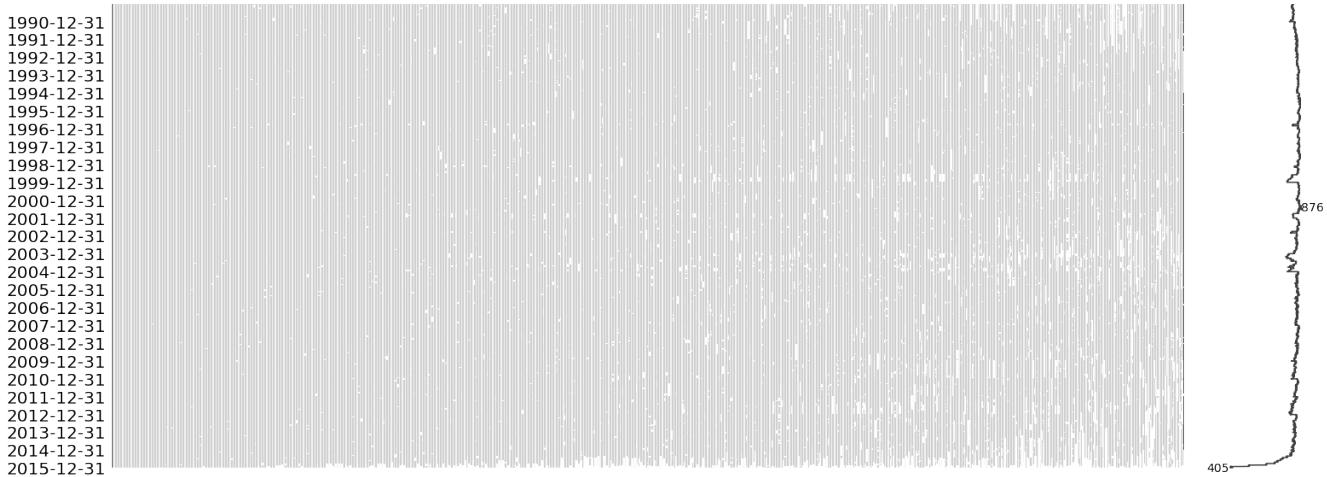


Figura 3.13: Espacios vacíos dentro del conjunto de datos sobre cada estación de precipitación.

Correlación entre precipitaciones y caudales Además de calcular las correlaciones entre los caudales, también exploramos cómo varía la correlación entre las precipitaciones y el caudal con respecto a la distancia. Tenemos la ubicación y la información de las precipitaciones en 878 estaciones diferentes durante el mismo período de tiempo que los caudales en Calamar. De esta forma, calculamos la correlación para cada una de estas estaciones frente su distancia a Calamar.

La gráfica de dispersión resultante en la figura 3.14 corrobora nuestra intuición en la que, en cuanto más lejos esté una estación de precipitación de la estación de Calamar, más independientes estadísticamente son la precipitación y el caudal.

Sin embargo, podemos observar que las mayores correlaciones están por debajo de un valor de 0,2. Si bien las precipitaciones pueden tener un efecto no despreciable sobre el caudal del río, éste puede depender en gran medida de otras variables climáticas, como el propio caudal en puntos vecinos aguas arriba del río, o también de como la temperatura media del país cambia, que a su vez está estrechamente relacionada con la ocurrencia de los fenómenos del Niño y la Niña. Por lo tanto, cuando estos fenómenos climáticos ocurren, hay un comportamiento atípico en los niveles de los ríos del país.

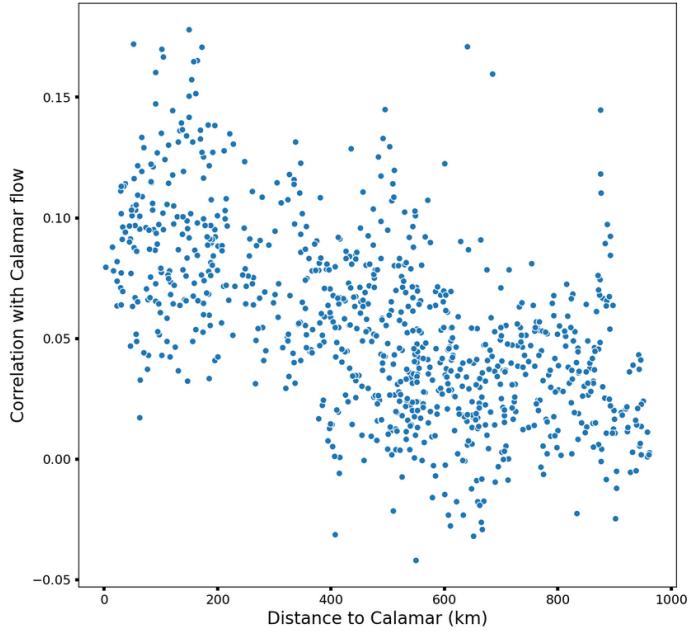


Figura 3.14: Correlación entre Calamar y la precipitación diaria según la distancia de las estaciones de medición.

3.4.2.4. Selección de datos

Después de realizar un análisis de los datos y ver que los datos de precipitación tienen una correlación muy baja con respecto a los caudales, se decidió no incluir estos datos al modelo. Aunque había poca correlación se desarrollaron los modelos a manera de prueba de concepto con el objetivo de conocer el rendimiento de los modelos precipitación-caudales, los modelos resultantes presentaban sobre-ajuste en todos los casos incluso al aplicar métodos de regularización. Lo anterior concuerda con Tanty y Desmukh (2015) donde se menciona que usualmente no se usa información de precipitación para la predicción de caudales.

3.4.3. Estrategia de trabajo

En esta sección proponemos dos modelos: Un modelo MLP, y un modelo de redes recurrentes, específicamente LSTM. Para esto definimos dos modelos base, y realizamos una afinación de hiper-parámetros por medio de experimentación.

Dividimos el conjunto de datos de los caudales en dos partes: el conjunto de entrenamiento que contiene los datos desde 1990 a 2010 (20 años) y el conjunto de prueba que contiene los datos desde 2011 a 2013 (3 años).

Para seleccionar el mejor modelo usamos la métrica de RMSE para evaluar los conjuntos de entrenamiento y pruebas. Esta métrica también nos sirve para verificar que tanto esta generalizando el modelo.

Para seleccionar la mejor relación entre la ventana de tiempo P mas grande y la longitud de la secuencia T mas pequeña, realizamos una comparación con los modelos elegidos, midiendo el desempeño de cada uno

los modelos en las diferentes configuraciones de P y T , en caso tal de que haya empate entre el rendimiento de las configuraciones P y T , se desempata con la métrica T/P (menor es mejor) esta métrica penaliza longitudes de secuencia largas y ventanas de tiempo cortas.

Se creó la métrica ζ , diseñada para que la diferencia entre el error de entrenamiento y el error de prueba sea mínimo (i.e. el modelo generaliza), especificado en la ecuación (3.10).

$$\zeta = 1 - \frac{\min(\epsilon_{\text{train}}, \epsilon_{\text{test}})}{\max(\epsilon_{\text{train}}, \epsilon_{\text{test}})} \quad (3.10)$$

donde ϵ_{train} y ϵ_{test} son el valor de la métrica del error para el conjunto de entrenamiento y pruebas respectivamente.

Idealmente esta métrica será $\zeta \approx 0$ si el modelo generaliza sobre ambos conjuntos de datos, acompañado de que ambos valores de error sean suficientemente bajos para el problema.

Para ambos modelos se estableció la métrica de **terminación temprana**, con *paciencia* de $p = 30$ épocas, vigilando el valor de error del conjunto de pruebas $\epsilon_{\mathcal{D}_{\text{test}}}$, si esta métrica no mejoraba después de p épocas, se terminaba la ejecución y se restauraban los parámetros del modelo para el cual la métrica fue mínima.

3.4.4. Modelo MLP

La entrada de la red está constituida por una secuencia de longitud n . Esta secuencia contiene n días de cada caudal anterior a la estación de Calamar (i.e. cinco caudales). La red recibe esta secuencia y pasa por cada una de las capas presentadas en la arquitectura. Las arquitecturas se presentan en el cuadro 3.2.

Las arquitecturas presentadas se realizaron por medio de experimentación siguiendo el siguiente proceso:

1. Proponer un modelo base de redes neuronales.
2. Agregar un número de capas densas aleatorias para determinar la profundidad de la red.
3. Determinar cuál el número de neuronas por capa.
4. Se regulariza el modelo de manera que el valor de error del conjunto de entrenamiento sea aproximado al conjunto de pruebas con regularización L^2 y *Batch Normalization*.
5. Afinamiento de hiper-parámetros como número de épocas, tasa de aprendizaje α , tamaño del *batch*, entre otros.

En todas las capas densas se establece una función de regularización L^2 con una tasa de regularización de $\lambda = 5 \times 10^{-3}$ para mitigar el sobre-ajuste durante el entrenamiento.

3.4.5. Modelo LSTM

La arquitectura de la red se obtuvo a partir de la experimentación con diferentes modelos, a continuación presentamos los modelos con el mejor desempeño. La arquitectura se presentan en el cuadro 3.3.

Tipo de capa	Función de activación	Neuronas
Input	—	$n \times 5$
Flatten	—	—
Densa	ELU	8
Batch Normalization	—	—
Densa	ELU	5
Densa	ELU	4
Densa	ELU	7
Densa	ELU	6
Densa	ELU	5
Densa	ELU	10
Densa	ELU	5
Output	ReLU	1

Cuadro 3.2: Arquitectura del modelo MLP.

Tipo de capa	Función de activación	Neuronas
Input	—	$n \times 5$
Densa	ELU	8
Batch Normalization	—	—
LSTM	Tanh	8
Densa	ELU	5
Densa	ELU	4
LSTM	Tanh	8
Densa	ELU	7
Densa	ELU	6
Densa	ELU	5
Densa	ELU	10
Densa	ELU	5
Output	ReLU	1

Cuadro 3.3: Arquitectura del modelo de la red recurrente.

3.5. Evaluación

3.5.1. Resultados

			LSTM			MLP		
T	P	T/P	τ	ϕ	ζ_{LSTM}	κ	ψ	ζ_{MLP}
120	7	17.14	1044.48	994.37	0.17	1727.09	2005.08	0.14
90	7	12.86	1748.11	1794.77	0.15	1090.73	633.52	0.42
60	7	8.57	3035.99	3162.85	0.29	1253.11	1550.61	0.19
30	7	4.29	1272.56	925.14	0.35	871.13	832.55	0.04
15	7	2.14	978.23	1022.94	0.23	980.28	791.14	0.19
7	7	1.0	1451.0	1307.94	0.22	1040.41	1072.91	0.03
120	15	8.0	2053.55	1716.58	0.27	1594.79	994.77	0.38
90	15	6.0	1884.14	2240.99	0.39	1366.35	733.87	0.46
60	15	4.0	1814.17	2072.41	0.11	1033.52	833.32	0.19
30	15	2.0	1052.74	952.74	0.24	1077.4	867.92	0.19
15	15	1.0	1267.44	960.74	0.14	1150.81	1234.47	0.07
7	15	0.47	1489.86	1355.18	0.34	1127.75	1019.98	0.1
120	30	4.0	2320.56	2154.6	0.11	1707.05	1561.35	0.09
90	30	3.0	1352.9	1348.81	0.2	1596.41	1232.51	0.23
60	30	2.0	1523.59	1866.13	0.1	1268.21	1267.02	0.0
30	30	1.0	1982.28	1843.43	0.21	1285.64	1255.46	0.02
15	30	0.5	1234.59	1377.25	0.15	1423.45	1225.51	0.14
7	30	0.23	1795.69	1635.74	0.22	1327.96	1189.32	0.1
120	45	2.67	1828.79	2143.33	0.04	1709.56	1736.94	0.02
90	45	2.0	2288.51	2190.43	0.01	1553.1	1603.05	0.03
60	45	1.33	2006.96	2122.8	0.15	1671.73	1904.14	0.12
30	45	0.67	3675.48	3906.27	0.06	1500.8	1908.97	0.21
15	45	0.33	1792.02	1988.35	0.04	1775.96	1669.74	0.06
7	45	0.16	1739.81	1828.75	0.12	1493.95	1837.07	0.19
120	60	2.0	1823.96	2175.8	0.01	1548.41	1786.63	0.13
90	60	1.5	1981.22	2239.72	0.03	1501.68	1770.74	0.15
60	60	1.0	2276.88	2211.99	0.06	1881.71	2023.82	0.07
30	60	0.5	1876.65	2098.26	0.03	1596.85	2075.06	0.23
15	60	0.25	1997.53	2123.96	0.24	1781.3	2067.93	0.14
7	60	0.12	2186.01	2106.82	0.18	1822.8	2051.34	0.11

Cuadro 3.4: Experimentos realizados sobre las arquitecturas de LSTM y MLP. T es la cantidad de días previos por cada caudal, P es la ventana de tiempo a predecir, $\text{RMSE}_{\text{train}}$ es el error RMSE (Root Mean Squared Error) sobre el conjunto de entrenamiento y $\text{RMSE}_{\text{test}}$ es el error RMSE sobre el conjunto de pruebas. τ y ϕ son los errores de RMSE de entrenamiento y pruebas respectivamente para la arquitectura con capas LSTM. κ y ψ son los errores de RMSE de entrenamiento y pruebas respectivamente para la arquitectura MLP. ζ es el nivel de generalización que se tiene por cada arquitectura.

Como se observa en el cuadro 3.4 se realizaron los experimentos respectivos para validar cuales son los mejores hiper-parámetros para obtener la mínima cantidad de días y la máxima ventana de predicción sin sacrificar el rendimiento del modelo. En el cuadro observamos que hay dos valores de rendimiento muy cercanos (30, 7) y (30, 15) para la red LSTM y (15, 7) y (30, 15) que para la MLP.

Los experimentos se realizaron con la arquitectura escogida en el cuadro 3.3.

Como se observa en las figuras 3.15 y 3.16 la simulación del modelo se adapta bastante bien a los datos

de prueba, se puede observar que el modelo aprendió la temporalidad y tendencia de los datos.

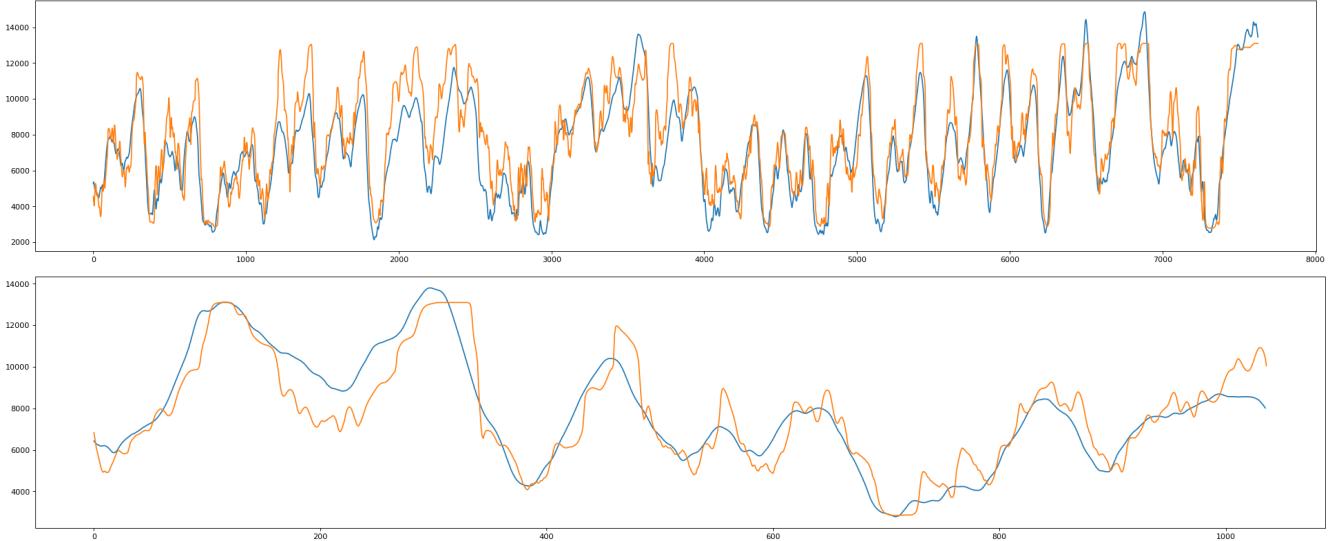


Figura 3.15: Gráficas de rendimiento del modelo LSTM escogido (30, 15) sobre el conjunto de entrenamiento (arriba) y el conjunto de prueba (abajo). La gráfica azul corresponde a los valores reales y la gráfica naranja son los valores predichos con base a la secuencia de valores de entrada.

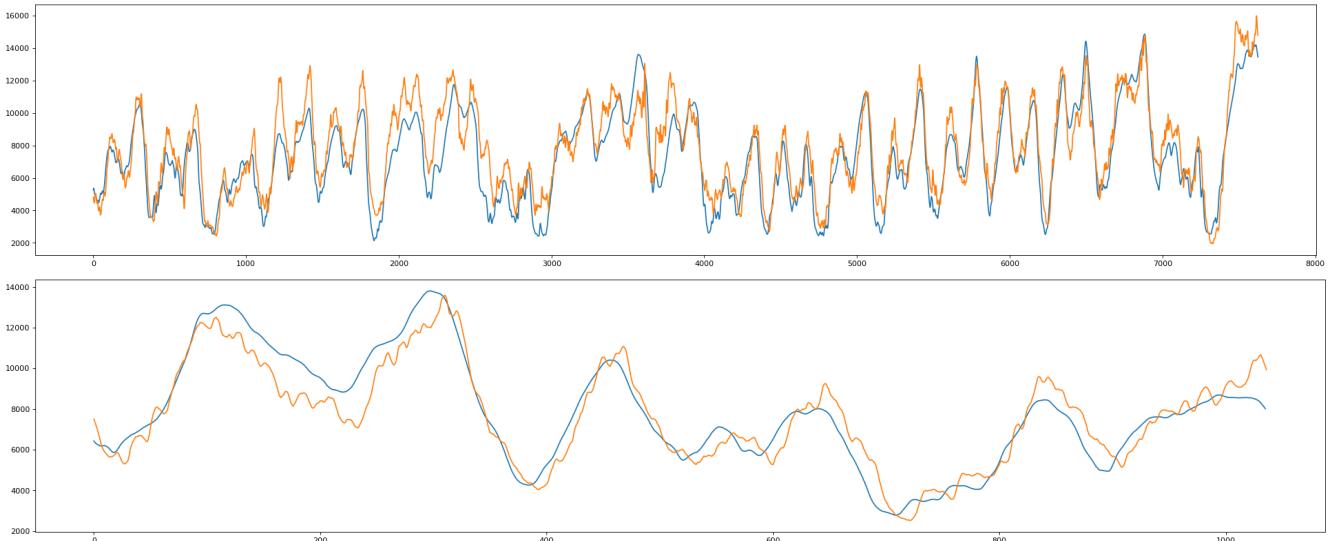


Figura 3.16: Gráficas de rendimiento del modelo MLP escogido (30, 15) sobre el conjunto de entrenamiento (arriba) y el conjunto de prueba (abajo). La gráfica azul corresponde a los valores reales y la gráfica naranja son los valores predichos con base a la secuencia de valores de entrada.

Como se puede observar en las figuras 3.15 y 3.16 y en el cuadro 3.4, ambos modelos presentan un

comportamiento similar, con la diferencia de que el modelo MLP presenta aparentemente menor ruido.

3.5.2. Hallazgos

Al momento de realizar el entrenamiento la ventana de tiempo de 120 días provoca que la gradiente en la red LSTM sea muy pequeña, esto debido a que las redes recurrentes sufren de una condición llamada gradiente desvaneciente y gradiente explosiva por su componente recurrente, y por tanto en muchos casos fue necesario aumentar la paciencia de la terminación temprana del modelo y la cantidad de épocas de manera que convergiera.

Al aplicar una ventana de promedio deslizante permite suavizar la curva de la serie de tiempo, donde adicionalmente reduce la presencia del componente del ruido aleatorio (apartado 3.2.2), esto permite eliminar las dificultades por la sensibilidad que el modelo pueda tener sobre cambios pequeños de la serie.

3.6. Conclusiones

Con la predicción de caudales con redes neuronales demostrada en este artículo, se puede concluir que este tipo de tecnologías pueden ser usadas para problemas de esta naturaleza. Otra conclusión obtenida durante la sección de evaluación fue que observamos que las redes MLP lograron ser más eficientes que las redes LSTM en este tipo de tareas.

El modelo TRAMO es una buena herramienta para completar los datos faltantes de las series de los caudales, lastimosamente tiene una capacidad limitada de 600 datos en el software, por lo que cuando hay vacíos grandes hay que realizar el mismo procedimiento varias veces.

Los datos de precipitación al ser tantos y tan dispersos, no son tan confiables y que al tener una correlación con los caudales tan baja, generaban modelos con poca capacidad de generalización al provocar sobre-ajuste sobre el conjunto de entrenamiento.

La metodología y la métrica usada para comparar las configuraciones de ventanas de tiempo con respecto a la longitud de la secuencia son sencillas y claras en su objetivo, por lo que con esta métrica se puede obtener la configuración que otorgue el mayor beneficio al modelo.

3.7. Trabajo futuro

Se podrían alimentar el modelo con nuevas variables como el nivel del río diario, variables macroclimáticas como las usadas en (Rojo-Hernández y Carvajal-Serna, 2010). También se podrían incluir datos extraídos de APIs públicas que ofrecen datos meteorológicos pre-procesados por modelos estadísticos especializados.

Explorar la posibilidad de utilizar auto-codificadores que transformen el espacio de entrada de los datos de precipitación a un espacio latente considerablemente reducido, de manera que solo las características relevantes sean codificadas, donde además el ruido se vería reducido.

Capítulo 4

Predicción del caudal de Calamar del Río Magdalena utilizando un framework orientado a objetos

Resumen

Las herramientas de desarrollo y los casos de estudio son recursos de aprendizaje esenciales cuando se están aprendiendo conceptos relacionados a las redes neuronales. Debido a esto en un trabajo anterior se desarrolló un *framework* que permite implementar soluciones, experimentar alternativas y extender su funcionalidad de manera sencilla, conociendo conceptos básicos de la programación orientada a objetos. El objetivo de este *framework* es que los interesados en este campo lo usen para incorporar mejor a los conceptos en el área.

En este trabajo se desarrolló un caso de estudio para el *framework* que, además de ser un valioso recurso de aprendizaje, nos permitió validar su fiabilidad, extensibilidad y facilidad de uso. Para esto replicamos una red neuronal, para la predicción de caudales del río Magdalena en Colombia, desarrollada en otro trabajo anterior e implementada en una biblioteca profesional como Keras.

El *framework* se extendió para lograr una implementación equivalente a la implementación profesional realizada en Keras. Adicionalmente, se realizó la comparación y análisis de los resultados de los modelos desarrollados en las dos herramientas.

Finalmente, se concluye que, aunque el *framework* no tiene el mismo desempeño que Keras, cumple su objetivo de ser una herramienta de aprendizaje, con un desempeño adecuado.

4.1. Introducción

Durante las últimas décadas el área de la inteligencia artificial ha tenido un crecimiento a pasos agigantados, esto se debe a los avances tecnológicos y a las implicaciones que las aplicaciones desarrolladas han tenido en nuestras vidas. Gran parte de estas aplicaciones están basadas en la técnica de las redes neuronales, por este motivo miles de empresas están contratando profesionales con conocimientos en esta área.

Durante este auge se han creado cientos de recursos (cursos virtuales, videos, blogs, etc) con los que se puede aprender este tipo de tecnología. Lastimosamente muchos de estos recursos se enfocan en mostrar como usar una herramienta y no en dar las bases teóricas para usarla, esto puede ocasionar problemas al momento de tomar una decisión en la arquitectura de la red. Otro problema encontrado en muchos de los recursos, es que la mayoría de estos tienden a escribir código sin considerar las buenas prácticas de programación, esto dificulta la lectura del código ocasionando que el desarrollador se pierda en el.

En diferentes trabajos se ha tratado de realizar herramientas para redes neuronales con el paradigma orientados a objetos; sin embargo, en gran parte de los trabajos solo cubren el perceptrón y el perceptrón multicapa dejando por fuera conceptos como las redes convolucionales y las recurrentes y las diferentes propuestas para las funciones de activación, las funciones de coste, los métodos de regularización, los métodos de optimización, entre otros.

Por este motivo desarrollamos un *framework* de redes neuronales que facilita el aprendizaje de los conceptos básicos de estas (Moreno Silva y Anzola Ávila, 2020). El aprendizaje se facilita debido a que el *framework* es muy sencillo de entender puesto que está hecho con programación orientada a objetos y con buenas prácticas de programación lo que facilita la lectura del diseño y del código fuente.

Adicionalmente, desarrollamos un modelo de redes neuronales que predice el caudal de la estación de Calamar, ubicada en el río Magdalena en Colombia (Anzola Ávila y Moreno Silva, 2020). En ese trabajo se compararon dos tipos de redes neuronales: las redes perceptron multicapa (MLP por sus siglas en inglés) y las redes Long-short Term Memory (LSTM). El resultado que se encontró fue que la red MLP fue más efectiva en este tipo de problemas frente a la LSTM.

En este trabajo replicamos el modelo de predicción de caudales con el *framework* de aprendizaje. La idea es ilustrar y validar lo sencillo que es pasar un modelo realizado en una biblioteca más avanzada como Keras al *framework* propuesto. Al final vamos a comparar las ventajas y las desventajas de cada una.

4.2. Marco teórico

4.2.1. *Framework*

Un *framework* es un artefacto de software que ofrece un entorno de trabajo y una serie de herramientas para resolver problemas en un dominio específico.

Existen varias bibliotecas conocidas para resolver problemas de aprendizaje automático, algunos de ellos son:

- **Keras** (Chollet *et al.*, 2015). Es una biblioteca de redes neuronales de código abierto, esta biblioteca tiene la peculiaridad de que puede ser usada con diferentes motores (como TensorFlow y Theano) y al día de hoy es la más usado por la comunidad.
- **TensorFlow** (Abadi *et al.*, 2015). Es una biblioteca de aprendizaje automático desarrollada por Google, especializada en redes neuronales. TensorFlow esta diseñado para ser muy eficiente debido a que esta muy bien optimizado para utilizar hardware de la maquina como las tarjetas de video y la concurrencia. A diferencia de Keras, TensorFlow tiene una curva de aprendizaje mucho más alta.
- **PyTorch** (Paszke *et al.*, 2019). Es una biblioteca de aprendizaje desarrollada por Facebook. Al igual que TensorFlow, PyTorch también esta optimizado para utilizar al máximo el hardware como la tarjeta gráfica y la concurrencia. PyTorch también ha absorbido otras bibliotecas, un ejemplo de eso fue Caffe2.
- **Theano** (Al-Rfou *et al.*, 2016). Este fue uno de las primeras bibliotecas de aprendizaje automático en nacer. Theano fue desarrollado por la Universidad de Montreal, este software fue descontinuado en 2019. En su versión final también era capaz de aprovechar el procesamiento por GPU y de manera concurrente.
- **Caffe** (Jia *et al.*, 2014). Esta biblioteca de aprendizaje automático fue desarrollada por la Universidad de Berkley, y al igual que Theano fue una de las primeras bibliotecas de aprendizaje automático en nacer, Caffe fue descontinuada en 2017, y basado en este salio Caffe2 que después fue absorbida por PyTorch.

El *framework* presentando en Moreno Silva y Anzola Ávila (2020) es desarrollado en el paradigma orientado a objetos siguiendo buenas prácticas de programación, esto con el objetivo principal de que las personas lo puedan modificar a conveniencia sin necesidad de conocer los detalles internos de la implementación.

Cabe aclarar que el *framework* después de ser extendido y modificado, puede ser usado como las librerías anteriormente mencionadas.

Este *framework* tiene 2 tipos de redes implementadas, las redes neuronales convolucionales (CNN por sus siglas en inglés) y las redes MLP. El *framework* también cuenta con la mayor parte de las funciones de activación, funciones de coste, métodos de regularización y de optimización. Adicionalmente también tiene un componentes de métricas y callbacks que pueden ser usados para obtener información relevante del modelo como la gráfica de la función de perdida mientras se ejecuta el entrenamiento.

Cabe aclarar que este *framework* esta orientado al aprendizaje de conceptos de redes neuronales, por lo que no debería ser usado en proyectos reales. Las soluciones se pueden experimentar en esta herramienta pero tendría una rendimiento inferior al usar librerías como TensorFlow ó PyTorch.

4.2.2. Modelos hidrológicos

El historial de caudales del Río Magdalena consta de realizar observaciones a diferentes estaciones ubicadas sobre la cuenca, cada una tomada en un intervalo de tiempo fijo.

El Instituto de Hidrología, Meteorología y Estudios Ambientales (IDEAM) de Colombia realiza mediciones regularmente de diferentes variables referentes al río. Este ha realizado estas mediciones por más de 50 años, hasta el día de hoy.

Todo este conjunto de datos recopilado por el IDEAM se compone en una secuencia temporal de valores por día correspondientes al caudal (o flujo) diario de varias estaciones presentes en la cuenca del río.

Existen múltiples investigaciones que tratan de modelar las dinámicas de los ríos y con esto predecir los comportamientos futuros que estos tendrán.

En particular Anzola Ávila y Moreno Silva (2020) presenta dos arquitecturas de redes neuronales, una red neuronal recurrente Long-Short Term Memory (LSTM) y una arquitectura de perceptrón multicapa (MLP por sus siglas en inglés). Ambas con el objetivo de predecir los caudales de la una estación llamada Calamar a partir de una secuencia de tiempo de los caudales adyacentes a esta estación, para después realizar una estimación del valor del caudal en una ventana de tiempo mayor a un día en el futuro.

4.3. Trabajos relacionados

4.3.1. Frameworks orientados a objetos

Existen muy pocas publicaciones que tienen aproximaciones orientadas a objetos para poder representar modelos de redes neuronales. En esta sección se presentan las aproximaciones encontradas.

En Caicedo Torres (2010) se realiza la implementación de una librería de redes neuronales para facilitar su enseñanza que incluye los conceptos de perceptrón, perceptrón multicapa, neurona, y algunos tipos de redes no supervisadas. Sin embargo el modelo presentado es muy básico, no se logran apreciar muchos componentes básicos de una red neuronal entre esos están: la función de costo, la función de activación, métodos de optimización, entre otros. A su vez este modelo tampoco es extensible, no es posible agregar nuevos conceptos debido a que no es clara la responsabilidad de cada clase.

En Valentini y Masulli (2002) se realiza una aproximación orientada a objetos con *C++*, sin embargo, no usan un lenguaje de modelado estándar por lo que comprender el diseño del autor se puede dificultar considerablemente. El autor incluye conceptos como algoritmos de gradiente descendiente, los conceptos básicos de la red y aparente mente algunos métodos usados dentro de la red. En este artículo también observamos que faltan muchos elementos, como la función de costo, la función de activación, entre otras.

En Ellingsen (1995) es uno de los mejores modelos que encontramos, ya que cuenta con un nivel claridad, y extensibilidad muy bueno. En este artículo encontramos conceptos como la neuronas y una especie de función de activación para cada una de estas, también se observan componentes como las capas y la red en si. Usan diagramas similares a UML lo que lo hace fácil de entender, sin embargo dada la edad

del articulo, faltan muchos conceptos que se han ido desarrollando como los métodos de optimización, los métodos de regularización entre otros.

Aunque Zaitsev (2017) no es un trabajo formal, tiene un modelo que cubre el perceptron multi capa (o *MLP* por sus siglas en inglés), y funciones de activación que son posibles de extender, por otra parte no se realizo lo mismo con las funciones de costo, y al igual que el trabajo anterior Ellingsen (1995) le faltan varios conceptos que se han ido desarrollando.

4.3.2. Modelos hidrológicos con redes neuronales

La mayor parte de las investigaciones sobre cuerpos hidrológicos intentan predecir sus dinámicas por medio de métodos estadísticos (Aguilar Villena, 2016; Rojo-Hernández y Carvajal-Serna, 2010; Amaris y Guerrero, 2017). Sin embargo, trabajos relacionados con redes neuronales para modelar estas dinámicas son mas comunes recientemente con la popularidad en auge del aprendizaje de maquina.

Uno de los varios trabajos en esta área con aprendizaje de maquina esta Veintimilla-Reyes y Cisneros (2015), quienes muestran un modelo de redes neuronales que intenta predecir el caudal a partir de datos meteorológicos como la precipitación. Por desgracia con este modelo no hay evidencia que la red que plantearon tenga una buena generalización al no incluir un conjunto de pruebas y tener un numero de iteraciones tan elevado para una red así de simple y un conjunto de datos tan pequeño.

Otro trabajo se presenta en Espinoza *et al.* (2018) se realiza la comparación de un modelo estadísticos junto con una arquitectura simple de red neuronal, sin embargo la arquitectura planteada contiene muy pocos parámetros debido a que la red presentada solo contiene solo 3 valores de entrada, 2 dos valores en una capa oculta y una sola salida. Tampoco especifican las entradas y las salidas de la red, por lo que no es posible saber que datos de entrenamiento usaron. A su vez no se evidencian métricas que confirmen el desempeño descrito.

Finalmente, en Tanty y Desmukh (2015) muestran distintos casos de uso en los que las redes neuronales pueden servir para resolver problemas de hidrología como escorrentía de drenaje, predicción de caudales, calidad del agua y modelos de agua subterránea. Con respecto a la tarea de estimación del caudal recomienda tomar como entrada los mismos caudales, y usualmente excluyendo variables como la precipitación.

4.4. Solución

4.4.1. Especificación del Problema

En este trabajo replicamos la arquitectura de la mejor solución implementada en Keras, con nuestro *framework*. El propósito es comparar la facilidad de la implementación, el rendimiento y el tiempo de entrenamiento.

4.4.2. Estrategia de trabajo

El conjunto de datos de los caudales los dividimos en dos partes: el conjunto de entrenamiento que contiene los datos desde 1990 a 2010 (20 años) y el conjunto de prueba que contiene los datos desde 2011

a 2013 (3 años). Efectuamos los mismos procedimientos de limpieza y pre-procesamiento del trabajo anterior.

Para comparar los modelos usamos la métrica de RMSE en la evaluación de los conjuntos de entrenamiento y pruebas. Esta métrica también nos sirve para verificar que tanto esta generalizando el modelo.

Para realizar la comparación los modelos en la dos herramientas, vamos a seguir el siguiente procedimiento:

1. Realizar la implementación de ambos modelos y comparar el código necesario.
2. Comparar los tiempos de entrenamientos tomando como base los hiper-parámetros definidos.
3. Verificar el rendimiento de cada modelo con los datos de entrenamiento y de validación.

4.4.3. Diseño

La mejor arquitectura para predecir el caudal de Calamar (Anzola Ávila y Moreno Silva, 2020), está ilustrada en el cuadro 4.1.

Tipo de capa	Función de activación	Neuronas
Input	—	$n \times 5$
Flatten	—	—
Densa	ELU	8
Densa	ELU	5
Densa	ELU	4
Densa	ELU	7
Densa	ELU	6
Densa	ELU	5
Densa	ELU	10
Densa	ELU	5
Output	ReLU	1

Cuadro 4.1: Arquitectura del modelo.

La entrada de la red esta constituida por una secuencia de longitud T . Esta secuencia contiene T días de cada caudal anterior a la estación de Calamar (i.e. cinco caudales). La red recibe esta secuencia y pasa por cada una de las capas presentadas en la arquitectura.

En todas las capas densas se establece una función de regularización L^2 con una tasa de regularización de $\lambda = 5 \times 10^{-3}$ para mitigar el sobre-ajuste durante el entrenamiento como es descrito en Anzola Ávila y Moreno Silva (2020).

4.4.4. Implementación de los modelos en los respectivos *frameworks*

A continuación se presentan las implementaciones de los respectivos *frameworks* a la arquitectura propuesta.

4.4.4.1. Keras

En el código 4.1 presentamos la implementación que realizamos en Keras sobre el modelo propuesto. En este código se observa que lo primero que hacemos es definir la arquitectura de la red, seguido de la función de perdida, y el optimizador, para después compilar y entrenar el modelo.

```

1 REG_RATE=0.005
2 ACTIVATION_FUNCTION = 'elu'
3
4 l2 = regularizers.l2(l2=REG_RATE)
5
6 x = inputs = keras.Input(shape=(SEQUENCE_LENGTH, 5))
7 x = layers.Flatten()(x)
8 x = layers.Dense(8, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
9 x = layers.BatchNormalization()(x)
10 x = layers.Dense(5, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
11 x = layers.Dense(4, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
12 x = layers.Dense(7, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
13 x = layers.Dense(6, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
14 x = layers.Dense(5, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
15 x = layers.Dense(10, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
16 x = layers.Dense(5, activation=ACTIVATION_FUNCTION, kernel_regularizer=l2)(x)
17
18 outputs = layers.Dense(1, activation='relu', kernel_regularizer=l2)(x)
19
20 model = keras.Model(inputs=inputs, outputs=outputs, name=name)
21
22 loss_f = tf.keras.losses.MeanSquaredError()
23
24 optimizer_f = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)
25
26 model.compile(
27     loss = loss_f,
28     optimizer = optimizer_f
29 )
30
31 history = model.fit(
32     x=train_sequence,
33     y=y_train_sequence,
34     batch_size = BATCH_SIZE,
35     epochs = EPOCHS,
36     validation_data = (validation_sequence, y_validation_sequence),
37     callbacks=[PlotLossesCallback()],
38     verbose=0)

```

Código 4.1: Implementación de una red neuronal en Keras.

4.4.4.2. Framework

Para realizar la implementación del modelo descrito en el cuadro 4.1, tuvimos que extender el *framework* para agregar la función de activación de ELU (*Exponential Linear Unit*) como se muestra en la figura 4.1.

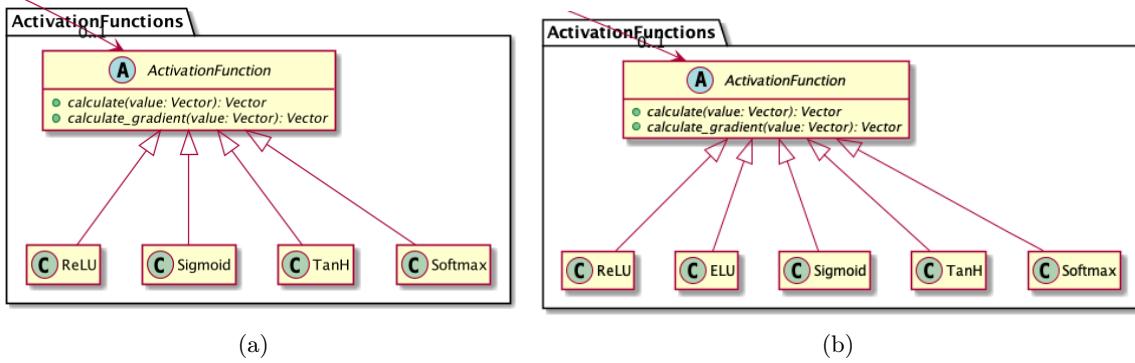


Figura 4.1: Extensión de las funciones de activación adicionando ELU. (a) Original y (b) extendido.

```

1  class ELU(ActivationFunction):
2
3      def __init__(self, alpha=1.0):
4          assert alpha >= 0.0, f"alpha should be greater than zero, got {alpha}"
5          self.alpha = alpha
6
7      def calculate(self, value: np.array) -> np.array:
8          return np.vectorize(lambda x: x if x >= 0 else self.alpha * (np.exp(x) - 1.))(value)
9
10     def calculate_gradient(self, value: np.array) -> np.array:
11         return np.vectorize(lambda x: 1 if x >= 0 else self.alpha * np.exp(x))(value)

```

Código 4.2: Implementación de la función de activación ELU.

Como se observa en el código 4.2, la implementación y la extensión del *framework* se puede hacer de una manera sencilla, en este caso observamos la implementación únicamente de dos funciones, en `calculate` se define la función de activación, y en `calculate_gradient` su gradiente.

Después de haber extendido el *framework*, se realiza la implementación del modelo descrito en el cuadro 4.1, el código referente a esa implementación se muestra en el código 4.3. El código inicia con la definición de la arquitectura, seguido de la función de coste, la función de optimización y las métricas a usar en el modelo.

```

1  self.architecture = [
2      ClassicLayer(8,    ELU(),    Xavier()),
3      ClassicLayer(5,    ELU(),    Xavier()),
4      ClassicLayer(4,    ELU(),    Xavier()),
5      ClassicLayer(7,    ELU(),    Xavier()),
6      ClassicLayer(6,    ELU(),    Xavier()),
7      ClassicLayer(5,    ELU(),    Xavier()),
8      ClassicLayer(10,   ELU(),   Xavier()),
9      ClassicLayer(5,    ELU(),    Xavier()),
10     ClassicLayer(1,    Relu(),   Xavier())
11 ]
12
13 self.cost_function = MeanSquaredError()
14 self.optimizer = Adam()
15
16 self.metrics = [MseMetric()]

```

```

17 self.callbacks = [GraphicCallback()]
18 self.learning_rate = 0.01
19 self.iterations = 30
20 self.batch_function = MiniBatch(self.training_data, self.expected_output, 256)
21
22 self.neural_net = NeuralNetwork(self.architecture,
23                                 self.cost_function,
24                                 self.learning_rate,
25                                 self.iterations,
26                                 optimizer=self.optimizer,
27                                 metrics=self.metrics,
28                                 callbacks=self.callbacks,
29                                 regularization_function = L2WeightDecay(0.005)
30 )
31
32 self.neural_net.train(self.training_data, self.expected_output, self.batch_function,
33                       self.validation_data)

```

Código 4.3: Implementación de una red neuronal en nuestro *framework*.

4.5. Evaluación

En esta sección vamos a comparar el comportamiento de las redes presentadas. Para eso vamos a tomar los hiper-parámetros y la configuración temporal (tamaño de la venta de tiempo y tamaño de la secuencia) que se obtuvieron durante el desarrollo en Keras.

4.5.1. Resultados

A nivel de implementación comparando el código 4.1 realizado en Keras, y el código 4.3 realizado en el *framework*, la definición de la arquitectura de la red es muy similar entre ellos, como se ve de la líneas 6..18 en Keras y en las líneas 1..10 en el *framework*. Keras por defecto usa Xavier como método de inicialización, y el *framework* usa el mismo método de regularización para todas las capas, en este caso se define en el constructor de la red.

Hay una diferencia en la definición de ambos modelos, puesto que Keras tiene un método que compila el modelo (método `compile`), mientras que el *framework* se le pasa a su constructor los elementos como la función de coste y el optimizador.

En Keras nos encontramos con el método `fit` el cual entrena el modelo con los datos de entrenamiento y lo prueba con los datos de validación durante el entrenamiento, y en el *framework* nos encontramos con el método de tren, donde únicamente ingresamos los datos de entrenamiento y de validación.

Framework				Keras			
T	P	RMSE _{train}	RMSE _{test}	$\zeta_{\text{Framework}}$	TI _{train}	RMSE _{train}	RMSE _{test}
		τ	ϕ			κ	ψ
30	15	1067.09	1492.72	0.29	125s	1077.4	867.92
						0.19	10s

Cuadro 4.2: Experimentos realizados sobre el *framework* y Keras.

En el cuadro 4.2 se observan el rendimiento que tienen los modelos tanto con los datos de entrenamiento como con los datos de pruebas, se tomó la configuración de (30, 15) puesto que es la que mejor comportamiento. T es la cantidad de días previos por cada caudal, P es la ventana de tiempo a predecir. $\text{RMSE}_{\text{train}}$ es el error RMSE (Root Mean Squared Error) sobre el conjunto de entrenamiento y $\text{RMSE}_{\text{test}}$ es el error RMSE sobre el conjunto de pruebas. τ y ϕ son los errores de RMSE de entrenamiento y pruebas respectivamente para la arquitectura del *framework*. κ y ψ son los errores de RMSE de entrenamiento y pruebas respectivamente para la arquitectura de Keras. ζ es el nivel de generalización que se tiene por cada arquitectura. TI_{train} es el tiempo que tomo el entrenamiento de los modelos en Keras y el *framework* respectivamente.

Observando los resultados obtenidos en el cuadro 4.2 notamos los siguientes puntos:

- **Rendimiento de entrenamiento:** En el *framework* su rendimiento respecto al error del conjunto de entrenamiento es 0,96 % mejor que el error de Keras. Siendo esto una diferencia poco significativa entre los dos.
- **Rendimiento de pruebas:** En Keras su desempeño tuvo una rendimiento 42 % mejor que el *framework*. Siendo Keras muy superior en desempeño. Una posible causa de esta diferencia el manejo de estabilidad numérica, debido a que en el *framework* no se realiza una comprobación constante para mantener los valores de los cálculos estables¹.
- **Tiempo de entrenamiento:** En este aspecto hubo una diferencia importante ya que al tiempo de entrenamiento del *framework* propuesto es un 92 % más lento en comparación a Keras. La causa es que Keras usa TensorFlow, el cual aprovecha los recursos de hardware al máximo, esto lo logra debido a que gran parte de esta librería esta escrita en lenguajes de programación de bajo nivel como C, C++ y CUDA C++.

En este cuadro podemos observar que Keras tiene un mejor rendimiento y demuestra un nivel de generalización superior, cabe aclarar que ninguno de los dos modelos llegó al punto de realizar sobreajuste.

Como se observa en las figuras 4.2 y 4.3 la simulación del modelo se adapta bastante bien a los datos de prueba, se puede observar que el modelo aprendió la temporalidad y tendencia de los datos.

¹Un calculo se considera estable si no sufre de *overflow* ó *underflow*.

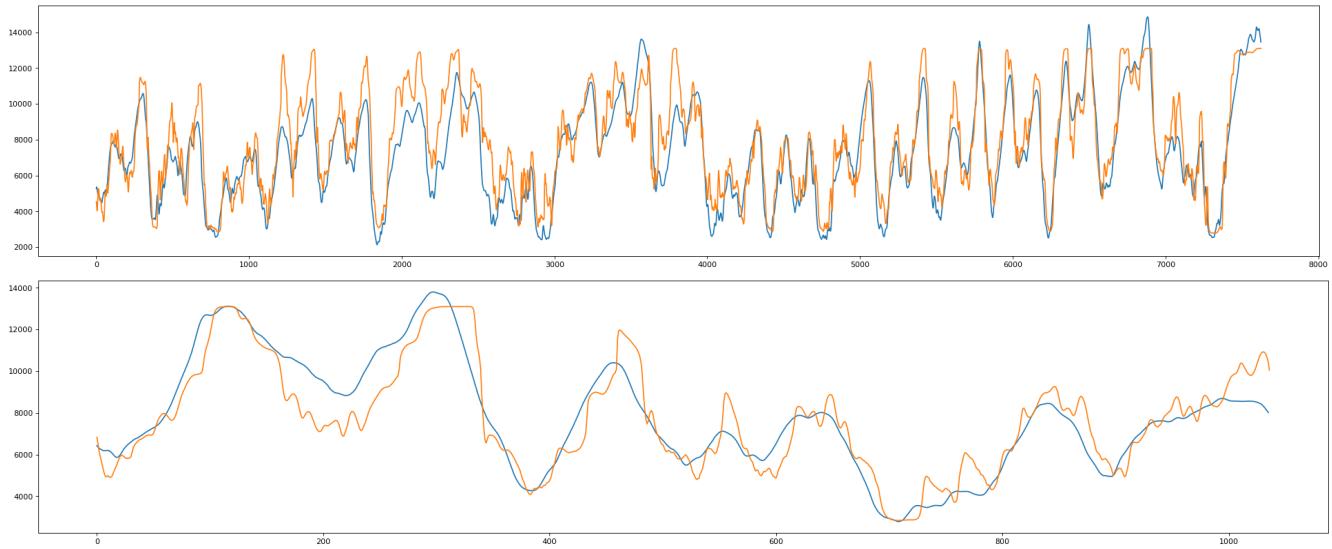


Figura 4.2: Gráficas de rendimiento del modelo realizado en Keras sobre el conjunto de entrenamiento (arriba) y el conjunto de prueba (abajo). La gráfica azul corresponde a los valores reales y la gráfica naranja son los valores predichos con base a la secuencia de valores de entrada.

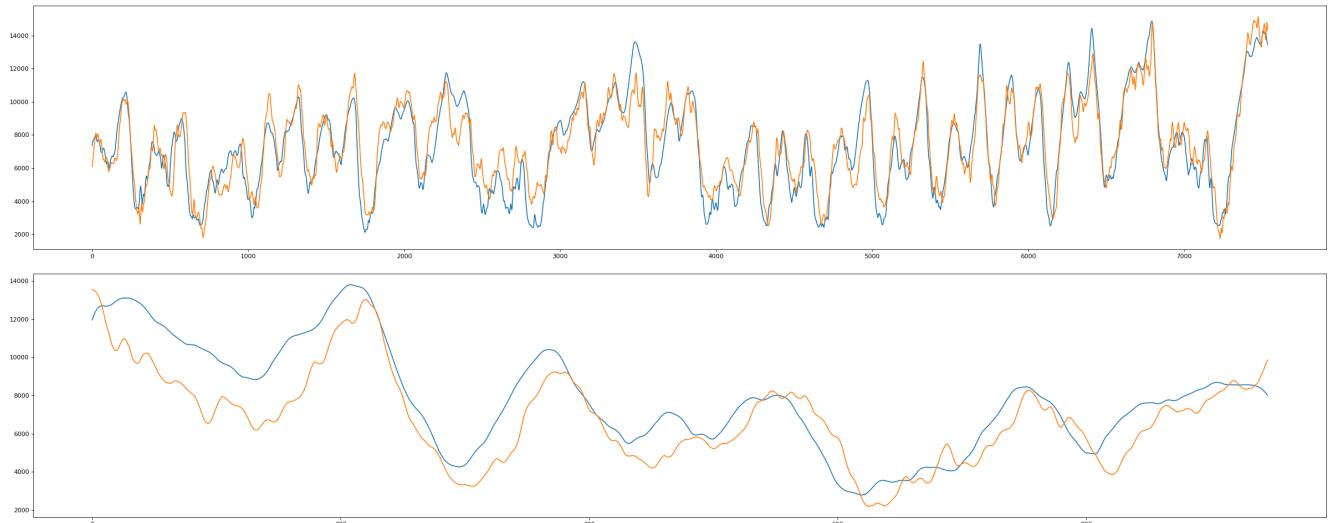


Figura 4.3: Gráficas de rendimiento del modelo realizado en el *framework* sobre el conjunto de entrenamiento (arriba) y el conjunto de prueba (abajo). La gráfica azul corresponde a los valores reales y la gráfica naranja son los valores predichos con base a la secuencia de valores de entrada.

Como se puede observar en las figuras 4.2 y 4.3 y en el cuadro 4.2, ambos modelos logran aprender la temporalidad y la tendencia de la serie, sin embargo el modelo del *framework* resultó ser un poco más pesimista y tiene un pequeño retraso con respecto al modelo de Keras.

4.5.2. Hallazgos

El *framework* fue capaz de simular el caudal con un nivel aceptable para ser un software diseñado para la enseñanza de modelos de aprendizaje automático.

Aunque el tiempo de entrenamiento del *framework* es aceptable hay que destacar que el modelo de Keras presenta una serie de optimizaciones que mejoran mucho los tiempos de entrenamiento. En particular, Keras utiliza internamente TensorFlow.

El modelo del *framework* presenta una gráfica mas limpia, sin embargo presenta un retraso con respecto al modelo de Keras.

4.6. Conclusiones

El río Magdalena es el recurso hídrico mas importante de Colombia, al ser capaces de predecir como este se comportará en el futuro, con poca incertidumbre, se pueden solucionar problemas claves con su manejo.

Existen múltiples soluciones que resuelven este problema con un grado de incertidumbre bastante bajo; sin embargo, soluciones que utilizan modelos de redes neuronales fallaban en utilizar metodologías del aprendizaje automático a su nivel fundamental.

En estudios anteriores se demostró que es posible realizar predicciones con un nivel de incertidumbre relativamente bajo con redes neuronales al ser estas aplicadas de manera apropiada con soluciones de software comercial.

Así mismo las herramientas de software usadas para la enseñanza de conceptos de aprendizaje automático, son un elemento valioso, para personas nuevas en este campo. En especial la herramienta de software que se uso, tuvo la flexibilidad de adaptarse al problema debido a su extensibilidad.

Basado en los resultados obtenidos se demostró que es posible usar este tipo de herramientas para resolver problemas en hidrología y que se puede llegar a tener comportamientos similares a herramientas comerciales con un nivel de error relativamente bajo.

4.7. Trabajo futuro

Como trabajo futuro vamos a desarrollar en el *framework* la red RNN con LSTM para compararla con la implementación de Keras. La comparación estará enfocada a ver la diferencia que existe en rendimiento, y destacar lo difícil o sencillo que puede resultar la implementación de las mismas.

Apéndice A

Notación

Números y Arreglos

a	Un escalar (entero o real)
\mathbf{a}	Un vector
\mathbf{A}	Una matriz
\mathbf{A}	Un tensor
I_n	Matriz identidad con n filas y n columnas
\mathbf{I}	Matriz identidad con dimensionalidad implícita por el contexto
\mathbf{a}	Una variable aleatoria real
\mathbf{a}	Un vector de variables aleatorias
\mathbf{A}	Una matriz de variables aleatorias

Conjuntos

\mathbb{A}	Un conjunto
\mathbb{R}	El conjunto de números reales
\mathbb{N}	El conjunto de números naturales
$[a, b]$	El intervalo real que incluye a y b
$[a : b]$	El intervalo discreto que incluye a y b
$(a, b]$	El intervalo real que no incluye a pero si b

Índices

a_i	Elemento i del vector \mathbf{a} , con índice empezando en 1
$A_{i,j}$	Elemento i, j de la matriz \mathbf{A}
$\mathbf{A}_{i,:}$	Fila i de la matriz \mathbf{A}
$\mathbf{A}_{:,i}$	Columna i de la matriz \mathbf{A}
$A_{i,j,k}$	Elemento (i, j, k) de un tensor \mathbf{A} 3-D
$\mathbf{A}_{:,:,i}$	Corte 2-D de un tensor 3-D
\mathbf{a}_i	Elemento i del vector aleatorio \mathbf{a}

Operaciones de Álgebra Lineal

\mathbf{A}^\top	Traspuesta de la matriz \mathbf{A}
$\mathbf{A} \odot \mathbf{B}$	Producto Hadamard (elemento por elemento) de \mathbf{A} y \mathbf{B} , las dimensiones de ambas es igual
$\det(\mathbf{A})$	Determinante de la matriz \mathbf{A}

Calculo

$\frac{dy}{dx}$	Derivada de y con respecto a x
$\frac{\partial y}{\partial x}$	Derivada parcial de y con respecto a x
$\nabla_{\mathbf{x}} y$	Gradiente de y con respecto a \mathbf{x}
$\nabla_{\mathbf{X}} y$	Matriz de derivadas de y con respecto a \mathbf{X}
$\nabla_{\mathbf{X}} y$	Tensor que contiene las derivadas de y con respecto a \mathbf{X}
$\nabla_{\mathbf{X}} y$	Tensor que contiene derivadas de y con respecto a \mathbf{X}
$\frac{\partial f}{\partial \mathbf{x}}$	Matriz Jacobiana $\mathbf{J} \in \mathbb{R}^{m \times n}$ de $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ para cualquier m y n

Teoría de Probabilidad e Información

$P(a)$	Una distribución de probabilidad de una variable aleatoria discreta
$p(a)$	Una distribución de probabilidad de una variable aleatoria continua
$a \sim P$	La variable aleatoria a tiene distribución P
$\text{Var}(f(x)) ; \sigma_{f(x)}^2$	Varianza de $f(x)$ sobre $P(x)$
$\text{Cov}(f(x), g(x))$	Covarianza de $f(x)$ y $g(x)$ sobre $P(x)$
$\mathcal{N}(\mathbf{x}; \mu, \sigma^2)$	Distribución gaussiana sobre \mathbf{x} con media μ y varianza σ^2

Funciones

$f : \mathbb{A} \rightarrow \mathbb{B}$	La función f con dominio \mathbb{A} y rango \mathbb{B}
$f \circ g$	Composición de las funciones f y g
$f(\mathbf{x}; \boldsymbol{\theta})$	Una función de \mathbf{x} parametrizada por $\boldsymbol{\theta}$.
$\log x$	Logaritmo natural de x
$\log_b x$	Logaritmo base b de x
$\ \mathbf{x}\ _p$	Norma L^p de \mathbf{x} , $\ \mathbf{x}\ _p = \sqrt[p]{\sum_i x_i^p}$
$\ \mathbf{x}\ _\infty$	Norma L^∞ de \mathbf{x} , $\ \mathbf{x}\ _\infty = \max_i x_i $
$\ \mathbf{x}\ _1$	Norma L^1 de \mathbf{x}
$\ \mathbf{x}\ = \ \mathbf{x}\ _2$	Norma L^2 de \mathbf{x} . También llamada norma ó distancia euclidiana.
$\mathbf{I} * \mathbf{K}$	Operación de convolución de una imagen \mathbf{I} y un filtro \mathbf{K}

Conjuntos de datos

m	Numero de ejemplares del conjunto de datos \mathcal{D}
\mathcal{D}	Conjunto de datos
$\mathcal{D}_{\text{train}}$	Conjunto de muestras de entrenamiento $\mathcal{D}_{\text{train}} \subset \mathcal{D}$
$\mathcal{D}_{\text{valid}}$	Conjunto de muestras de validación $\mathcal{D}_{\text{valid}} \subset \mathcal{D}$
$\mathcal{D}_{\text{test}}$	Conjunto de muestras de pruebas $\mathcal{D}_{\text{test}} \subset \mathcal{D}$
(\mathbf{X}, \mathbf{Y})	Matrices de entrada y salida con m filas de ejemplos
\mathbf{X}	La matriz de $m \times n$ con muestra de entrada $\mathbf{x}^{(i)}$ en la fila $\mathbf{X}_{i,:}$
$\mathbf{x}^{(i)}$	La i -ésima muestra (entrada) de un conjunto de datos de entrada \mathbf{X}
$\mathbf{y}^{(i)}$	El objetivo asociado con $\mathbf{x}^{(i)}$ para aprendizaje supervisado
$\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(t)}$	Una secuencia de t elementos

Redes neuronales

l	Numero de capas de toda la red
$n^{[i]}$	Numero de neuronas de la capa i -ésima
$\theta ; \theta^{[i]}$	Parámetros de todo el modelo ; parámetros de la capa i -ésima de la red, su estructura dependerá del tipo de capa en el contexto
$\mathbf{W}^{[i]}$	Matriz de pesos de la capa i -ésima
$\mathbf{b}^{[i]}$	Sesgo de la capa i -ésima
$\mathbf{z} = \mathbf{W}^{[i]} \mathbf{x} + \mathbf{b}^{[i]}$	Logits de la capa i -ésima con entrada $\mathbf{x} \in \mathbb{R}^{1 \times n^{[i-1]}}$ de la capa anterior, pesos $\mathbf{W} \in \mathbb{R}^{n^{[i-1]} \times n^{[i]}}$ y sesgos $\mathbf{b} \in \mathbb{R}^{n^{[i]} \times 1}$
$g(\mathbf{z})^{[i]}$	Función de activación de la capa i -ésima
$\mathbf{x}^{(i)}, \mathbf{y}^{(i)}$	Entrada \mathbf{x} y salida esperada \mathbf{y} del ejemplo i -ésimo
$\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}$	Salida esperada y salida estimada del ejemplo i -ésimo
J	Función de costo

Series de tiempo

$x_{\{t\}}$	Valor de una serie de tiempo en el instante t
$x_{\{a\}}, \dots, x_{\{b\}}$	Secuencia de valores de una serie de tiempo desde el instante a al instante b

Índice de figuras

2.1.	Red perceptron multicapa (MLP)	9
2.2.	Modelo sugerido por Caicedo Torres (2010).	13
2.3.	Aproximación en Valentini y Masulli (2002).	13
2.4.	Aproximación en Ellingsen (1995).	14
2.5.	Aproximación en Zaitsev (2017).	14
2.6.	Diagrama de clases de la red neuronal con sus propiedades y métodos.	15
2.7.	Diagrama de las clases <code>NeuralNetwork</code> y <code>Layer</code>	16
2.8.	Diagrama de clases sobre la clase <code>Layer</code>	17
2.9.	Diagrama simplificado del tipo de clases que existen con relación de herencia a la clase <code>Layer</code>	18
2.10.	Diagrama de la clase <code>ActivationFunction</code> de las funciones de activación con sus diferentes implementaciones y su relación con la clase <code>Layer</code>	19
2.11.	Diagrama de clases de las funciones de coste con sus diferentes implementaciones.	21
2.12.	Funciones para obtener Batches.	23
2.13.	Diagrama de clases de los métodos de regularización con sus diferentes implementaciones.	25
2.14.	Diagrama de clases de los métodos de optimización con sus diferentes implementaciones. .	26
2.15.	Diagrama de clases de las funciones de inicialización con sus diferentes implementaciones.	29
2.16.	Diagrama de clase de redes neuronales densas clásicas <code>ClassicLayer</code>	30
2.17.	Diagrama de clases de las capas del paquete de convolución.	31
2.18.	Diagrama de clases de las capas del paquete de redes neuronales recurrentes.	32
2.19.	Diagrama de las clases <code>Callback</code> y <code>Metric</code>	34
2.20.	Modelo completo.	40
3.1.	Ubicaciones de las estaciones.	44
3.2.	Ejemplo de una serie de tiempo	45
3.3.	Red perceptrón multicapa (MLP)	48
3.4.	Red neuronal recurrente (RNN)	49

3.5.	Red Long-Short Term Memory (LSTM)	49
3.6.	Flujo del <i>cell state</i> en una red LSTM	49
3.7.	Red Gated Recurrent Unit (GRU)	50
3.8.	Distribución de espacios vacíos dentro del conjunto de datos sobre cada estación de caudal.	52
3.9.	Matriz de correlación entre caudales de las estaciones.	53
3.10.	Caudales de la estación de Calamar.	54
3.11.	Caudales de la estación de Calamar, eje adyacente con un solo año.	54
3.12.	Serie de tiempo de una estación con ruido, y después de aplicar el pre-procesamiento de datos con una ventana de promedio deslizante de 15 días ($w = 15$).	55
3.13.	Espacios vacíos dentro del conjunto de datos sobre cada estación de precipitación.	56
3.14.	Correlación entre Calamar y la precipitación diaria según la distancia de las estaciones de medición.	57
3.15.	Gráficas de rendimiento del modelo LSTM escogido	61
3.16.	Gráficas de rendimiento del modelo MLP escogido	61
4.1.	Extensión de las funciones de activación adicionando ELU	70
4.2.	Gráficas de rendimiento del modelo realizado en Keras	73
4.3.	Gráficas de rendimiento del modelo realizado en el <i>framework</i>	73

Índice de cuadros

2.1.	Configuraciones de los operadores AND, NAND y OR.	10
2.2.	Funciones de costo	21
2.3.	Comparación de rendimiento entre arquitecturas	37
2.4.	Ejemplos de clasificación binaria de texto.	37
3.1.	Porcentaje de valores no-nulos respecto al total de los datos.	52
3.2.	Arquitectura del modelo MLP.	59
3.3.	Arquitectura del modelo de la red recurrente.	59
3.4.	Experimentos realizados sobre las arquitecturas de LSTM y el MLP	60
4.1.	Arquitectura del modelo.	68
4.2.	Experimentos realizados sobre el <i>framework</i> y Keras.	71

Referencias

- Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., y Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.
- Aguilar Villena, R. (2016). Prediccion de caudales en el rio Chira con fines de descolmatacion del embalse de poechos.
- Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Bleeker Snyder, J., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P.-L., Cho, K., Chorowski, J., Christiano, P., Cooijmans, T., Côté, M.-A., Côté, M., Courville, A., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Ebrahimi Kahou, S., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I., Graham, M., Gulcehre, C., Hamel, P., Harlouchet, I., Heng, J.-P., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrancois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P.-A., Mastropietro, O., McGibbon, R. T., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, E., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., y Zhang, Y. (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, **abs/1605.02688**.
- Amaris, G. y Guerrero, T. (2017). Applying ARIMA model for annual volume time series of the Magdalena River. *Tecnura*, **21**(52), 88–101.
- Anzola Ávila, A. y Moreno Silva, J. A. (2020). Predicción del caudal calamar del río magdalena con redes neuronales. Escuela Colombiana de Ingeniería Julio Garavito.
- Baiocchi, G. y Distaso, W. (2003). Gretl: Econometric software for the gnu generation. *Journal of Applied Econometrics*, **18**(1), 105–110.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.
- Breiman, L. (1996). Bagging predictors. *Machine Learning*, **24**, 123–140.
- Briñez, Á. P. y Nieto, F. H. (2005). Ajuste de un modelo no lineal a la variable precipitación en una estación hidro-meteorológica de Colombia Fitting a nonlinear model to the precipitation variable in a Colombian hydrological/meteorological station. *Revista Colombiana de Estadística*, **28**(2), 113–124.
- Caicedo Torres, W. (2010). Diseño e implementación de una librería neuronal y de una suit didáctica para la enseñanza sobre redes neuronales.
- Cauchy, A. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, **25**(1847), 536–538.
- Chollet, F. et al. (2015). Keras. <https://keras.io>.
- Christopher, O. (2015). Understanding LSTM Networks. <https://colah.github.io/posts/2015-08-Understanding-LSTMs/>.
- Chung, J., Çaglar Gülcöhre, Cho, K., y Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *ArXiv*, **abs/1412.3555**.

- Ellingsen, B. K. (1995). An object-oriented approach to neural networks.
- Escuela de Administración, Finanzas e Instituto Tecnológico (2020). ¿Por qué se desbordan los ríos? - Escuela de Ciencias / Noticias - Universidad EAFIT. <https://www.eafit.edu.co/escuelas/ciencias/noticias/Paginas/por-que-se-desbordan-los-rios.aspx>. [Online; accessed 26. Jul. 2020].
- Espinoza, J., Campos, J., Lopez, N., y Ardiles, Y. (2018). Predicción de caudales aplicando modelo de escorrentía de deshielo (SRM), y redes neuronales artificiales para la cuenca del Río Huasco, región de Atacama, Chile. *Vallenar*, (November).
- Fernández, N., Jaimes, W., y Altamiranda, E. (2010). Neuro-fuzzy modeling for level prediction for the navigation sector on the Magdalena River (Colombia). *Journal of Hydroinformatics*, **12**(1), 36–50.
- Glorot, X. y Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research*, **9**, 249–256.
- Goodfellow, I., Bengio, Y., y Courville, A. (2016). *Deep Learning*. MIT Press. <http://www.deeplearningbook.org>.
- He, K., Zhang, X., Ren, S., y Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034.
- Hochreiter, S. y Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, **9**, 1735–1780.
- Holmes, E. E., Scheuerell, M. D., y Ward, E. J. (2020). *Applied Time Series Analysis for Fisheries and Environmental Sciences*.
- Huang, J., Jin, P., Xiang, J., Li, L., Jiang, X., y Wei, C. (2015). Application of grey prediction and BP neural network in hydrologic prediction. *MATEC Web of Conferences*, **25**, 1–7.
- Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, **9**(3), 90–95.
- Ioffe, S. y Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *ArXiv*, [abs/1502.03167](https://arxiv.org/abs/1502.03167).
- Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., y Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*.
- Katanforoosh, K. y Kunin, D. (2019). Initializing neural networks. <https://www.deeplearning.ai/ai-notes/initialization/>.
- LeCun, Y., Cortes, C., y Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, **2**.
- Maravall, A. y Peña, D. (1996). Missing observations and additive outliers in time series models.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition.
- Moreno Silva, J. A. y Anzola Ávila, A. (2020). Redes neuronales: Una aproximación orientada a objetos. Escuela Colombiana de Ingeniería Julio Garavito.
- Murphy, K. P. (2012). *Machine learning - a probabilistic perspective*.
- Oliphant, T. E. (2006). *A guide to NumPy*, volume 1. Trelgol Publishing USA.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., y Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, y R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.
- Pillai, A. (2009). Designing and implementing a neural network library for handwriting detection, image analysis. <https://www.codeproject.com/Articles/14342/Designing-And-Implementing-A-Neural-Network-Librar>.
- Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, **4**(5), 1–17.
- Rojo-Hernández, J. D. y Carvajal-Serna, L. F. (2010). Predicción no lineal de caudales utilizando variables macroclimáticas y análisis espectral singular. *Tecnología y Ciencias del Agua*, **1**(4), 59–73.
- Rumelhart, D. E., Hinton, G. E., y Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, **323**, 533–536.

- Seymour, L., Brockwell, P. J., y Davis, R. A. (1997). *Introduction to Time Series and Forecasting.*, volume 92.
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., y Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, **15**, 1929–1958.
- Tanty, R. y Desmukh, T. S. (2015). Application of Artificial Neural Network in Hydrology- A Review. *International Journal of Engineering Research and*, **V4**(06), 184–188.
- Tarapuez Roa, J. C. y Eslava Avendaño, D. E. (2011). TSW (TRAMO-SEATS), una aplicación para la emisión primaria en Colombia. <http://www.fce.unal.edu.co/unidad-de-informatica/proyectos-de-estudio/economia/1274-tsw-tramo-seats-3.html>. [Online; accessed 25. Jul. 2020].
- Valentini, G. y Masulli, F. (2002). NEUROObjects: An object-oriented library for neural network development. *Neurocomputing*, **48**, 623–646.
- Vargas Cuervo, G. y Peña, Y. (2014). *Caracterización Física y Análisis Sociocultural del Riesgo Asociado a las Inundaciones del Canal del Dique, Colombia*, page 18.
- Veintimilla-Reyes, J. y Cisneros, F. (2015). Predicción de Caudales Basados en Redes Neuronales Artificiales (RNA) para Períodos de Tiempo Sub Diarios. *Revista Politécnica*, **35**(3), 42–49.
- Wan, L., Zeiler, M. D., Zhang, S., LeCun, Y., y Fergus, R. (2013). Regularization of neural networks using dropconnect. In *ICML*.
- Zaitsev, O. (2017). Single-layer Perceptron in Pharo. Towards Data Science.
- Zhou, V. (2019). An Introduction to Recurrent Neural Networks for Beginners. <https://victorzhou.com/blog/intro-to-rnns>. [Online; accessed 1. May. 2020].