Escuela Politécnica Superior de Jaén
Máster Universitario en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Máster

Métodos de aprendizaje profundo para la predicción de series temporales

Alumno/a: Manuel Germán Morales

Tutor/a: Antonio Jesús Rivera Rivas
Cristóbal José Carmona del Jesus

Dpto: Informática

UNIVERSIDAD DE JAÉN

D. Antonio Jesús Rivera Rivas y D. Cristóbal José Carmona del Jesus, tutores del Trabajo Fin de Máster titulado: **Métodos de aprendizaje profundo para la predicción de series temporales**, que presenta Manuel Germán Morales, autorizan su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, enero de 2024

El estudiante

GERMAN MORALES MANUEL - 20622668V

Firmado digitalmente por GERMAN MORALES MANUEL - 20622668V

Manuel Germán Morales

Los tutores

RIVERA RIVAS ANTONIO JESUS - 26015863B

Firmado digitalmente por RIVERA RIVAS ANTONIO JESUS - 26015863B
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-26015863B, givenName=ANTONIO JESUS, sn=RIVERA RIVAS, cn=RIVERA RIVAS ANTONIO JESUS - 26015863B
Fecha: 2024.01.25 17:24:09 +01'00'

Antonio Jesús Rivera Rivas

CARMONA DEL JESUS CRISTOBAL JOSE - 75102603J

Firmado digitalmente por CARMONA DEL JESUS CRISTOBAL JOSE - 75102603J
Fecha: 2024.01.25 16:54:22 +01'00'

Cristóbal José Carmona del Jesus

Agradecimientos

Al igual que en el Trabajo de Fin de Grado, no voy a desaprovechar este espacio para dejar constancia de que todo esto ha sido posible a un grupo de personas que han estado ahí y me han ayudado a culminar estos estudios de máster de la mejor forma posible: haciendo algo que me encanta (aunque algunas veces me haya provocado pensar en abrir una panadería y dejar todo este mundo aparte). Cómo el trabajo trata sobre el aprendizaje, voy a intentar ser un poco más alternativo para que sea más interesante y conseguir que de una vez mis padres se enteren qué hago en mi trabajo al leerla.

Todos sabemos lo complejo que es este mundo. En nuestro día a día procesamos grandes cantidades de estímulos que nos llegan a saturar, e incluso somos capaces de tomar decisiones en situaciones nunca antes vistas gracias a la asombrosa capacidad de nuestro cerebro para aprender y adaptarse. Sin embargo, ***nadie nace sabiendo***, y es aquí donde entráis vosotros, los padres, responsables de *etiquetar* con mucho cariño nuestro mundo a lo largo de todas nuestras *épocas* para que logremos alcanzar lo mejor de nosotros. Gracias a vuestro *refuerzo* habéis conseguido que hoy llegue a estar otra vez aquí, así que me gustaría repetiros que muchas gracias por toda la *transferencia* (no bancaria, que también, pero de conocimiento) que habéis hecho y la que todavía queda.

También agradecer a otros *agentes* que llevan ayudándome a *ajustar mis parámetros* desde hace un tiempo: mis amigos y compañeros. Este año no voy a poner una lista de nombres como el anterior, porque todos sabéis quienes sois. Me habéis proporcionado ese *ajuste* necesario para seguir *clasificando* mis prioridades y demostrarme que mi *gradiente* aún puede llegar al máximo global que tanto soñamos.

Otra parte de mi *ajuste* es y sigue siendo influenciada, como no, por mis tutores: Antonio y Cristóbal. Y no solo ellos, sino que también me gustaría dedicar este párrafo a todo el grupo SiMiDat. Gracias por aguantar con paciencia mis *estancamientos* y guiarme hacia mi *óptimo*, demostrándome que soy capaz de manejar todos estos conceptos tan abstractos con seguridad. La finalización de este trabajo es solo el inicio de otro camino que comenzará en un futuro inmediato (en el cual espero que no se vuelva a romper ningún disco duro más).

Espero que disfrutéis este trabajo tanto como yo y que os resulte igual de interesante. Gracias por todo y, como cómicamente suelo decir, lo siento por tan poco.

Tabla de contenidos

Lista de figuras

Lista de tablas

Siglas

AGI *Artifcial General Intelligence*, Inteligencia Artificial General.

AI *Artificial Intelligence*, Inteligencia Artificial.

ANI *Artifcial Narrow Intelligence*, Inteligencia Artificial Específica.

ARIMA *AutoRegressive Integrated Moving Average*, Media Móvil Integrada Autorregresiva.

ASI *Artifcial Super Intelligence*, Super Inteligencia Artificial.

BPPT *BackPropagation Through Time*, Propagación hacia atrás a lo largo del Tiempo.

CNN *Convolutional Neural Network*, Red Neuronal Convolucional.

DL *Deep Learning*, Aprendizaje Profundo.

GAN *Generative Adversarial Network*, Red Generativa Adversaria.

GPU *Graphics Processing Unit*, Unidad de Procesamiento de Gráficos.

HPC *High Performance Computing*, Computación de Altas Prestaciones.

Leaky-ReLU *Leaky Rectified Linear Unit*, Unidad Lineal Rectificada con Fugas.

ML *Machine Learning*, Aprendizaje Máquina.

PReLU *Parametric Rectified Linear Unit*, Unidad Lineal Rectificada Paramétrica.

RELU *Rectified Linear Unit*, Unidad Lineal Rectificada.

RNN *Recurrent Neural Network*, Red Neuronal Recurrente.

SMAPE *Symmetric Mean Absolute Percentage Error*, Error Porcentual Absoluto Medio Simétrico.

STL *Seasonal and Trend decomposition using LOESS method*, descomposición de la tendencia estacional usando el método LOESS.

TCN *Root Mean Squared Error*, Error Cuadrático Medio Raíz.

TCN *Temporal Convolutional Network*, Red Convolucional Temporal.

TLU *Threshold Logic Unit*, Unidad Lógica de Umbral.

TSF *Time Series Forecasting*, Predicción de Series Temporales.

WAPE *Weighted Average Percentage Error*, Error Porcentual Medio Ponderado.

Capítulo 1

INTRODUCCIÓN

En este capítulo se presenta una primera sección donde se comentarán varios aspectos introductorios que justifican la realización de este trabajo de fin de máster. La segunda sección describe el problema a resolver desde un punto de vista más formal, con sus correspondientes objetivos. Por último, se presentará una planificación temporal del proyecto y una estimación de sus costes.

1.1. Motivación

En nuestro día a día existen una gran cantidad de fenómenos que presentan distintos comportamientos en función del momento en el que se observen. La medición de la temperatura del aire en una estación meteorológica, la ocupación de cualquier tipo de servicio, las pulsaciones de cualquier persona que posea una pulsera inteligente o las existencias de cualquier producto suministrado por cualquier comercio son ejemplos cuya actividad es influencia por la presencia de dependencias temporales fuertes o débiles. Siguiendo el ejemplo de la temperatura, es lógico que al medio día sea más alta que en la mañana o en la noche. Además, en función de la estación podemos asumir su evolución.

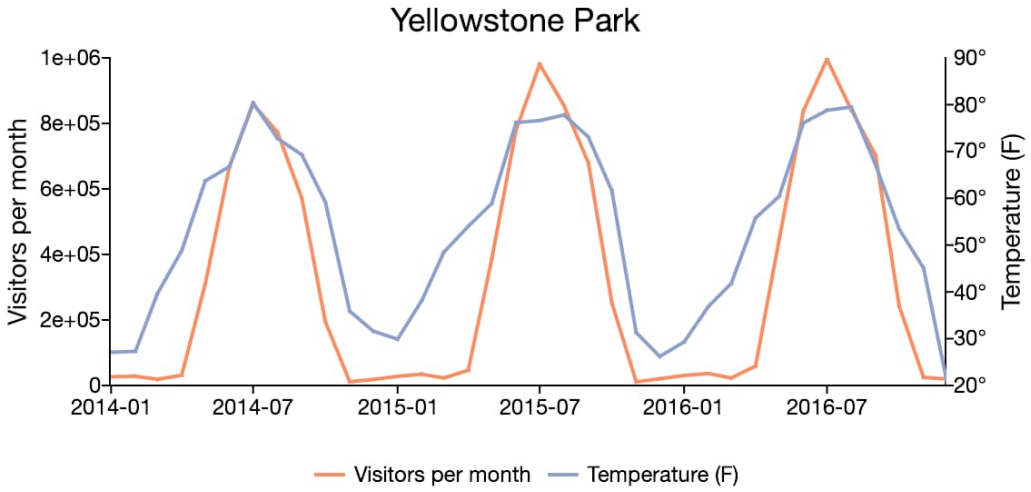

Figura 1.1: Dos series temporales que muestran la evolución de la temperatura y de los visitantes del parque Yellowstone entre los años 2014 y 2016. Podemos ver que en el mes de agosto (es decir, en verano), el número de visitantes llega a su máximo en todos los años. Fuente: Clower [2022].

A todas estas casuísticas cuyas mediciones son recogidas y están ordenadas a lo largo del tiempo se le denominan **series temporales**, como las que se pueden apreciar en la figura 1.1. Ellas proporcionan una fuente de información valiosa que, analizada correctamente, pueden informarnos de diferentes patrones (como tendencias o ciclos) que nos ayudarán a comprender mejor el comportamiento del dominio que estemos estudiando.

Por ello, podemos intuir que es una buena idea tener en cuenta no solo el valor de la observación, sino el momento temporal en el que se sitúa. Como punto de partida para abordar este problema, se comenzaron a aplicar técnicas estadísticas para determinar las predicciones. Posteriormente, la llegada de la Inteligencia Artificial permitió seguir avanzando en la resolución de esta tarea aplicando algoritmo de aprendizaje

supervisado y no supervisados, como un simple k-*Vecinos Cercanos* para estimar el valor futuro de la serie o agrupamientos de diferentes series mediante k-*medias* para detectar aquellas que sean anómalas.

Con el auge del aprendizaje automático, y especialmente el aprendizaje profundo, una gran cantidad de técnicas surgieron para intentar resolver las diferentes tareas que comprenden este tipo de dato. En este trabajo nos centraremos en el problema de la **predicción** de series temporales usando algoritmos de aprendizaje profundo, los cuales han demostrado ser adecuados gracias a su sobresaliente capacidad de generalización.

Encontrar técnicas que nos permitan inferir valores futuros de estas series es relevante por la gran cantidad de dominios que las contienen, como vimos en el primer párrafo. Áreas como la logística, la sanidad o la energía pueden beneficiarse del uso de estos modelos para, entre otros ámbitos, estimar la demanda de un producto, la propagación de enfermedades o el consumo eléctrico de los hogares. Evidentemente, no son los únicos campos que pueden utilizar este tipo de métodos para predecir el futuro y apoyar sus decisiones basándose en el conocimiento extraído a partir de los datos.

1.2. Objetivos del proyecto

El objetivo general del proyecto es comprender y estudiar el funcionamiento de los principales métodos usados para la predicción de series temporales basados en el paradigma de aprendizaje profundo. Para lograrlo, definiremos los siguientes objetivos específicos (OE).

- **OE1**: Estudio de los conceptos clave del campo de la Ciencia de Datos en general y del aprendizaje profundo en particular.
- **OE2**: Estudio del campo de la predicción de series temporales.
- **OE3**: Estudio de métodos de aprendizaje profundo para la predicción de series temporales.
- **OE4**: Adaptación y ejecución de métodos de aprendizaje profundo para la predicción de series temporales.
- **OE5**: Evaluar, comparar y analizar los resultados obtenidos.

1.3. Estructura de la memoria

Seguidamente, describiremos la estructura de esta memoria, que reflejará la completitud de los objetivos descritos en la sección anterior.

En el **capítulo 2** centraremos nuestra atención en los **antecedentes** del problema a resolver. Empezaremos introduciendo el contexto en el que se enmarca nuestro proyecto, hablando sobre qué es la Inteligencia Artificial, la Ciencia de Datos y el aprendizaje automático. Explicaremos el modelo de neurona usado en la mayoría de modelos profundos para posteriormente describir algunos de ellos. También enfatizaremos en la definición de serie temporal junto a sus propiedades más relevantes y finalizaremos describiendo cómo se ha abarcado su predicción con enfoques estadísticos y basados en aprendizaje profundo.

Tras conocer el contexto y los fundamentos de los modelos profundos destinados a la predicción de series temporales, detallaremos los **materiales y métodos** usados. Las herramientas usadas para su desarrollo, los conjuntos de datos seleccionados y su preprocesado y el marco de la experimentación a realizar se expondrán en el **capítulo 3**.

Una vez realizados los experimentos propuestos en el capítulo anterior, procederemos al análisis de sus **resultados** en el **capítulo 4**. Usaremos diferentes representaciones para mejorar la visibilidad de ellos.

Por último, finalizaremos este proyecto exponiendo las **conclusiones** obtenidas tras el análisis de resultados realizado en el **capítulo 5**. Además, se comentarán varias ideas de cara a la producción de un estudio más extenso.

1.4. Planificación y costes

Una vez comentada la motivación y principal finalidad del proyecto, se procede a realizar una planificación de las diferentes tareas que se requieren para la realización de este, así como una estimación de su coste.

1.4.1. Planificación temporal

En la figura 1.2 se refleja la asignación temporal de las tareas en las que ha sido dividido el proyecto. La planificación es orientativa y puede sufrir retrasos o modificaciones a medida que se avanza en un proyecto. Se han tomado 25 horas semanales de trabajo a 18.05€/hora (Talent.com [2024]). Es importante tener en cuenta que una celda de la cuadrícula se corresponde con media semana.

1.4.2. Estimación de costes temporales

En la tabla 1.1 se muestran los costes orientativos asociados a cada tarea nombrada en la planificación temporal para este trabajo experimental, permitiéndonos obtener una aproximación de su coste.

Tarea	Duración	Coste
Revisión bibliográfica	3 semanas	1353.75€
Selección y preprocesamiento de datos	1.5 semanas	676.88€
Implementación de modelos recurrentes	2 semanas	902.50€
Implementación de modelo convolucional	1 semana	451.25€
Revisión de la implementación	0.5 semanas	225.63€
Experimentación	4 semanas	1805.00€
Análisis de resultados	2 semanas	902.50€
TOTAL	**14 semanas**	**6317.51€**

Tabla. 1.1: Relación de costes temporales

Iniciaremos el proyecto centrando nuestra atención en la realización de una **revisión bibliográfica** centrada en el aprendizaje profundo aplicado a la predicción de series temporales, a partir de la cual seleccionaremos los modelos a implementar. Posteriormente, exploraremos diversas fuentes para **seleccionar los conjuntos de datos** a usar y determinaremos su **preprocesamiento**. Paralelamente a esta tarea empezaremos con el **desarrollo de los modelos profundos**. Tras procesar nuestros conjuntos de datos, implementar nuestros modelos y revisar que dicha implementación sea correcta, procederemos a **lanzar la experimentación** y, una vez finalizada, **analizaremos** los resultados obtenidos.

Identificaremos dos puntos críticos en este proyecto, que deben de ser completados a tiempo para su éxito: la **implementación de los modelos** y la **experimentación**. Cualquier desfase en las tres tareas que representan estos hitos supondrá un **retraso considerable** de la finalización del proyecto.

Ciertamente, no es muy común disponer de clúster HPC debido al auge de los servicios en la nube. Por ello, se ha decidido incluir en estos costes el precio de usar de una máquina virtual de altas prestaciones proporcionada por *Amazon Web Services* que proporciona la empresa *Amazon*. Por lo tanto, el coste de la tarea de **experimentación** se incrementa en **19342.98€** (Amazon [2024]) si deseamos usar una instancia de *Amazon EC2 P3* durante 4 semanas a 28,78€/hora. Existen alternativas más económicas, pero se ha seleccionado un producto que coincida con las características especificadas en 3.4.3.

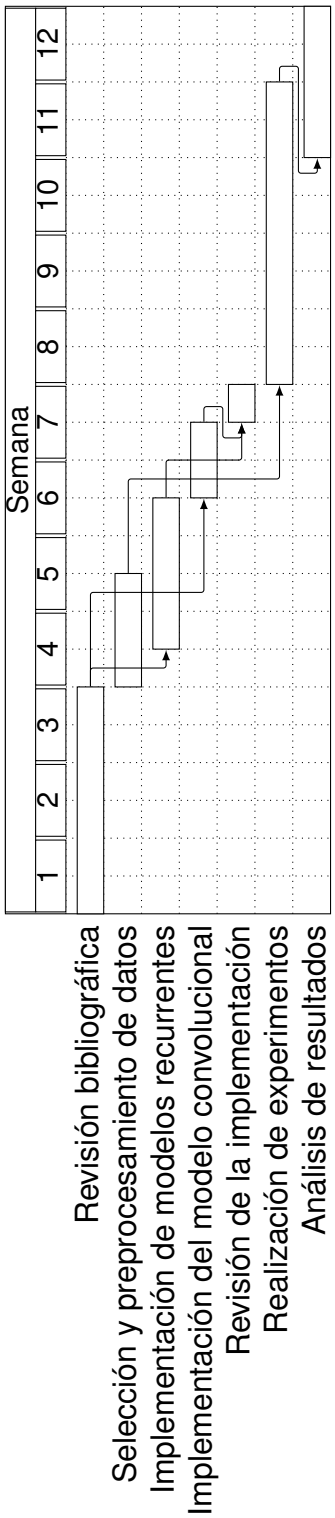

Figura 1.2: Diagrama de Gantt del proyecto. Fuente: Elaboración propia.

Capítulo 2

ANTECEDENTES

Para poder afrontar la tarea de predicción de series temporales es necesario estudiar el contexto que rodea a esta tarea. En la primera parte de este capítulo introduciremos conceptos generales (pero esenciales) que son necesarios para abarcar cualquier tarea relacionada con el DL. Posteriormente, nos centraremos en el ámbito de las series temporales hablando sobre sus características y finalizaremos explicando algunos métodos relevantes para su predicción.

2.1. Contexto

Antes de presentar la tarea de predicción de series temporales es conveniente introducir la disciplina y campos que la envuelven. En los siguientes puntos se detalla brevemente qué es la IA, la Ciencia de Datos, el ML y el DL junto a una serie de conceptos situados dentro de los mismos, necesarios para entender con claridad la tarea a resolver junto a sus métodos.

2.1.1. Inteligencia Artificial

A lo largo de la historia se han ido proponiendo diferentes definiciones de Inteligencia Artificial. Podemos decir que todas ellas comparten la idea de que se trata de una disciplina enmarcada dentro del ámbito de la Informática, cuyo objetivo es el desarrollo de máquinas que son capaces de tomar decisiones racionales en un entorno para poder lograr un objetivo. En 2019, el [1]Grupo de Alto Nivel en IA (AI-HLEG) nombrado por la Comisión Europea, emite un comunicado en el que define qué es y qué puede ser un sistema inteligente:

> "*Artificial intelligence (AI) refers to systems that display intelligent behaviour by analysing their environment and taking actions – with some degree of autonomy – to achieve specific goals. AI-based systems can be purely software-based, acting in the virtual world (e.g. voice assistants, image analysis software, search engines, speech and face recognition systems) or AI can be embedded in hardware devices (e.g. advanced robots, autonomous cars, drones or Internet of Things applications)*" - High-Level Expert Group on Artificial Intelligence [2019]

Detallar también que los métodos usados para reproducir esta inteligencia no tienen que limitarse a comportamientos observables en la naturaleza, tal y como comenta McCarthy [2004] en su definición: "*but AI does not have to confine itself to methods that are biologically observable*".

Es relevante definir los conceptos de IA general (o fuerte, AGI) y específica (o débil, ANI). En Abonamah et al. [2021] se especifica la principal diferencia entre ambos tipos de IA.

[1]AI-HLEG es un grupo de expertos en IA formado por la Comisión Europea cuyo objetivo es aconsejarla a la hora de desarrollar su estrategia respecto a esta disciplina. (Más información sobre AI-HLEG)

La **ANI** es solo capaz de realizar tareas específicas (ya sea solo una o un conjunto reducido) y es la más usada actualmente. Los sistemas diseñados bajo esta concepción de la IA poseerán un gran conocimiento sobre el dominio en el que operan, pudiendo resolver su tarea incluso a tiempo real. Sin embargo, estos sistemas quedarán completamente inútiles al cambiarlos de entorno, ya que no poseen conocimiento sobre este. Un claro ejemplo de ANI es el sistema de conducción autónoma de un coche inteligente. El sistema posee un gran conocimiento de todos los elementos contenidos en el código de circulación, pero este mismo no sería capaz de operar si se acopla a una aeronave debido al que el código de circulación aéreo es totalmente diferente.

Por otra parte, una **AGI** se caracteriza por ser más cercana a la inteligencia humana. Este tipo de AI debe de ser capaz de aprender y lograr afrontar cualquier tipo de tarea que un ser humano pueda realizar. Debido a que un sistema debe de presentar características como poseer consciencia de sí mismo para ser considerado AGI no se ha encontrado ninguna evidencia de que exista este tipo de AI, por lo que conseguirla se consideraría un gran logro en esta disciplina.

Bostrom [1998] define la existencia de una forma de AI que va mucho más allá que la AGI: la **ASI**. Esta inteligencia se define por ir más allá que la del hombre, siendo “mucho más inteligente que los mejores cerebros humanos en prácticamente todos los campos". Es imposible actualmente lograr el desarrollo de sistemas que cumplan con las características descritas por este tipo de inteligencia.

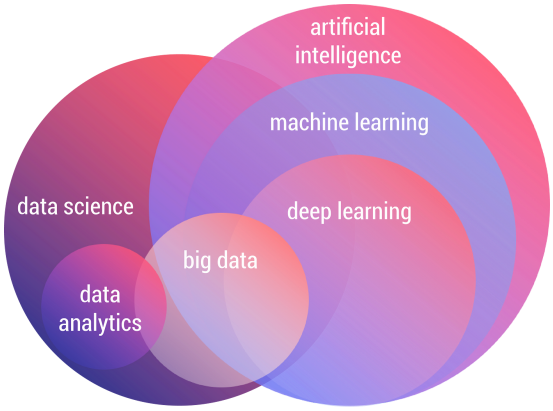

Figura 2.1: Situación de diferentes campos de la IA y cómo apoya a otros, como la Ciencia de Datos. Fuente: Yalçin [2020].

Esta gran disciplina se compone de una gran cantidad de campos. En este trabajo nos centraremos en el estudio e implementación de métodos basados en DL, el cual se sitúa dentro del ML. Además, los métodos y herramientas construidos bajo esta disciplina sirven de apoyo para otras tareas y áreas como la Ciencia de Datos (figura 2.1),

manifestando así su interdisciplinariedad y utilidad en nuestro día a día. Aclararemos en los siguientes apartados los de Ciencia de Datos, ML y DL.

2.1.2. Ciencia de Datos

El término "Ciencia de Datos" surgió a principios de los años sesenta para describir el campo de una nueva profesión que ayudaría a comprender e interpretar las grandes cantidades de datos que se estaban acumulando en aquel momento (Foote [2021]), aunque hasta 1974 no se clarifica qué es gracias a la definición dada por Peter Naur (Liguori [2020]) en su artículo "Concise Survey of Computer Methods". Al igual que sucede con IA, existen una gran variedad de puntos de vista que originan diferentes definiciones de Ciencia de Datos.

De todos los puntos de vista existentes, podemos destacar dos: el que define Ciencia de Datos según el objetivo general del campo y el que indica las cualidades debe de poseer aquella persona que trabaje en el nicho de este campo.

El primer punto de vista contiene definiciones como la mostrada en el artículo Rizk and Elragal [2020], en la que se habla de Ciencia de Datos como el proceso de extraer conocimiento a partir de los datos para apoyar así a otra tarea, como la toma de decisiones.

> "*La **Ciencia de Datos** es el estudio de la extracción sistemática de patrones no obvios y útiles y de los datos con el fin de avanzar en la investigación, tomar decisiones organizativas y facilitar una sociedad basada en datos.*"

Aunque como hemos mencionado, también podemos definir la Ciencia de Datos desde el punto de vista de lo que se espera de un científico de datos, poniendo de manifiesto los diferentes campos en los que poseen habilidades junto a la capacidad de utilizarlas bajo un enfoque holístico para lograr así completar sus tareas (Zhu and Xiong [2015]):

> "*La **Ciencia de Datos** es una integración de métodos estadísticos, tecnología informática e inteligencia artificial.*'

Dentro de la Ciencia de Datos también definimos un **proceso**, el cual nos guiará desde la recolección y procesamiento de los datos hasta la extracción de patrones y generación de conocimiento de los mismos. Este proceso de Ciencia de Datos se

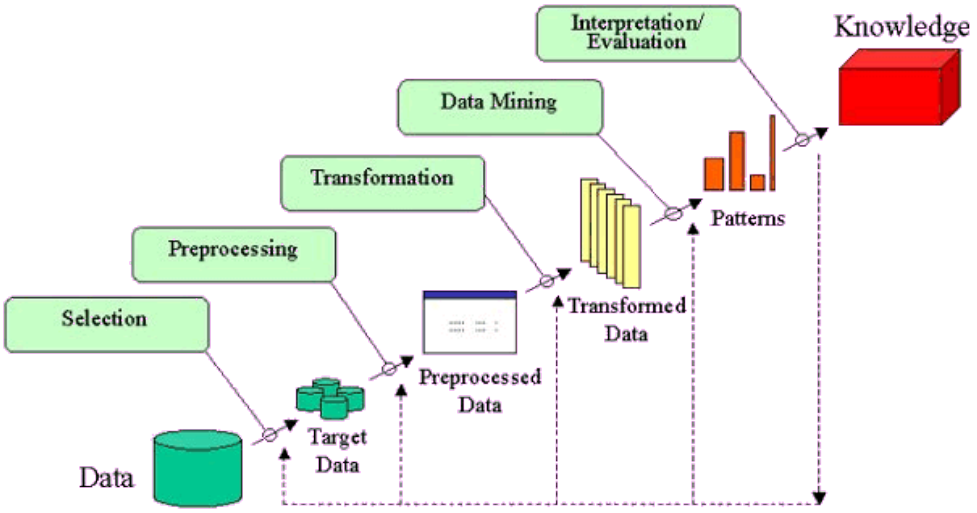

Figura 2.2: Proceso de Extracción del Conocimiento en Bases de datos. Fuente: Hamilton [2000].

conoce originalmente como Extracción del Conocimiento en Bases de datos (KDD, *Knowledge Discovery in Databases*) (Fayyad et al. [1996]) y cuenta con varias etapas, descritas a continuación (figura 2.2):

1. **Selección**: Determinar con qué subconjunto de datos almacenado en la base de datos queremos trabajar.

2. **Preprocesamiento**: Mejorar la calidad de los datos para obtener un mejor rendimiento a la hora de extraer conocimiento de ellos, aplicando una serie de procedimientos o estrategias que permitan lidiar con valores perdidos, ruido o anómalos.

3. **Transformación**: Una vez preprocesados los datos, podemos también transformarlos para mejorar aún más todavía su calidad. En esta etapa se puede realizar una selección de características relevantes en función de nuestro objetivo, así como una reducción o aumento de la dimensionalidad del conjunto de datos o cualquier transformación que creamos que sea conveniente (por ejemplo, una normalización).

4. **Minería de Datos**: La Minería de Datos se encarga de buscar los patrones y relaciones del conjunto de datos preparado y transformado. Existen una gran variedad de técnicas para ello, por lo que deberemos seleccionar el que más se adecue a nuestra tarea.

5. **Interpretación y evaluación**: Por último, debemos de interpretar las salidas del algoritmo anterior para obtener así el conocimiento deseado. Además, podemos evaluar los patrones y relaciones anteriores para determinar su calidad e ir iterando sobre todo el proceso para conseguir progresivamente mejores resultados, dando lugar a un conocimiento de mayor calidad.

Aclarar que actualmente, aunque se siga usando el término “base de datos” en el proceso, se refiere a cualquier documento capaz de almacenar información y que la Ciencia de Datos no se apoya únicamente en técnicas provenientes de la IA, tal y como se ilustra en la figura 2.1.

2.1.3. Aprendizaje Automático

Un tipo de algoritmos usados en el ámbito de la Ciencia de Datos para realizar la extracción de patrones son los basados en Aprendizaje Automático, los cuales son un subconjunto de la IA.

La característica diferenciadora de este tipo de algoritmos es la ausencia de una definición de reglas para la resolución de la tarea tratar, ya que lo que se busca es la extracción de los patrones y relaciones de los datos de entrada para generar así una abstracción que está intrínseca en el mismo algoritmo para posteriormente poder usarla como si de un conocimiento general se tratase (figura 2.3). A esta abstracción se le denomina **modelo** y al hecho de encontrar los patrones y relaciones de los datos de entrada se le denomina **aprender**.

Figura 2.3: Proceso de ML. Fuente: Chandramouli et al. [2018]

Podemos clasificar los algoritmos de ML en diferentes grupos en función de cómo realizan el aprendizaje. Normalmente, encontraremos tres tipos de aprendizaje: Supervisado, no supervisado y por refuerzo. Aunque también podemos encontrar otros tipos de aprendizaje híbridos, como el semi-supervisado.

Aprendizaje supervisado

En este tipo de aprendizaje, el objetivo del algoritmo es aprender la relación entre la entrada y la salida de un conjunto de datos. Este conjunto estará formado por pares, emparejando el conjunto de características de entrada (x_i) con una etiqueta (y_i), es por ello que a este tipo de conjuntos de datos se les denomina **etiquetados** (2.1).

$$D = \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\} \tag{2.1}$$

Las principales tareas situadas dentro del paradigma del aprendizaje supervisado son la clasificación, la regresión y la predicción de series temporales. Hablaremos brevemente de las dos primeras en los siguientes párrafos y comentaremos la última en las siguientes secciones.

El objetivo de la tarea de clasificación es encontrar una relación entre los datos de entrada y un conjunto de etiquetas discretas denominadas clases a partir de la abstracción aprendida por el algoritmo. En función del número de clases a asignar podemos diferenciar dos tipos de clasificaciones: binaria (dos clases) y multiclase (más de dos clases). Si se asigna más de una clase a los datos de entrada, estaremos hablando de clasificación multietiqueta.

En la tarea de regresión nos encontramos un objetivo muy similar al de la tarea anterior, con el único matiz de que las etiquetas son continuas y no discretas. La predicción de series temporales es un tipo de regresión la cual tiene en cuenta las dependencias temporales de los datos de entrada para predecir su evolución.

Aprendizaje no supervisado

A diferencia del aprendizaje supervisado, en esta metodología no se cuentan con datos etiquetados. El propio algoritmo es el encargado de analizar los datos de entradas para asignarles una etiqueta, formando así grupos de instancias similares. Debido a ello, este enfoque es muy adecuado para la tarea de agrupamiento o *clustering*.

Otra tarea a destacar es el descubrimiento de asociaciones, la cual busca encontrar relaciones entre las variables del conjunto de datos. Esta tarea es relevante en el análisis de transacciones y mercados, concretamente en el análisis de la cesta de la compra, ya que permite obtener qué productos se suelen comprar juntos (ejemplo: “Los clientes que compran X también suelen comprar Y.”).

Aprendizaje por refuerzo

Este tipo de aprendizaje es diferente a los dos anteriores: no requiere de un conjunto de datos para aprender porque su objetivo principal no es buscar las relaciones y patrones presentes en ellos, sino en un **entorno**, por lo que no es ni supervisado ni no supervisado.

En esta metodología de aprendizaje, el algoritmo será el encargado de determinar el **comportamiento del agente**, es decir, de decidir cuál es la mejor acción (de un conjunto finito) a realizar en función del estado actual del entorno. El estado actual se obtiene mediante una **observación** del entorno, la cual es recogida mediante sensores presentes en el mismo agente. El entorno puede ser modificado mediante los actuadores que él posee, dando lugar a un nuevo estado que será evaluado para determinar si el agente ha decidido correctamente.

Aprendizaje semi-supervisado

El paradigma de aprendizaje semi-supervisado usa conjuntos de datos que presentan una cantidad de instancias etiquetadas baja respecto al número de instancias no etiquetadas, debido al alto coste o dificultad que supone etiquetarlas. Es por ello que el principal propósito de este tipo de aprendizaje es estudiar cómo podemos aumentar el número de ejemplos etiquetados aprovechando los datos no etiquetados, ya que se encuentran disponibles sin ningún coste adicional.

En líneas generales, este tipo de paradigma emplea modelos supervisados o no supervisados para asignar a los ejemplos no etiquetados aquella etiqueta que presente mayor confianza a partir de las instancias etiquetadas, a esta se le denomina pseudoetiqueta. En la figura 2.4 se muestra este proceso, permitiendo así tener un conjunto de datos compatible con cualquier algoritmo de aprendizaje supervisado gracias a la inclusión de etiquetas.

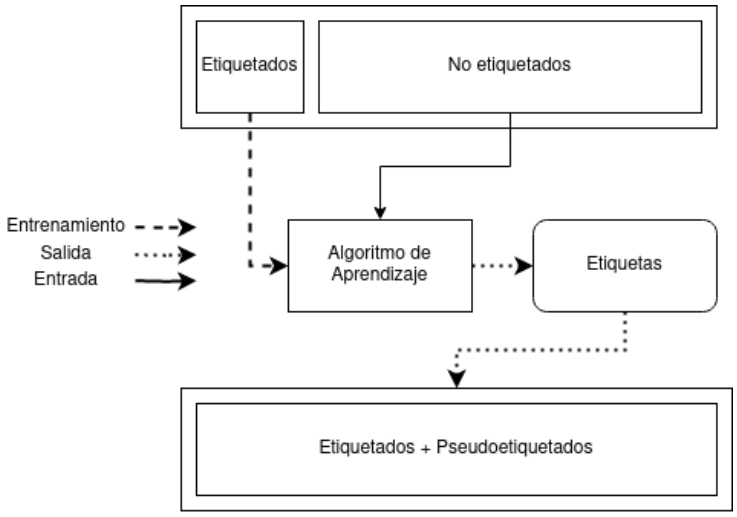

Figura 2.4: Proceso de aprendizaje semi-supervisado. Fuente: Elaboración propia.

2.1.4. Aprendizaje Profundo

El Aprendizaje Profundo (DL) está formado por un subconjunto de técnicas situadas dentro del ML (figura 2.5) basadas en la composición de diversas funciones lineales y no lineales que son aplicadas a los datos de entrada para conseguir varias representaciones, cada vez más abstractas, de los datos de entrada permitiendo así el aprendizaje de otras funciones altamente complejas (Lecun et al. [2015]). Las **redes neuronales** aplican la metodología descrita anteriormente para realizar su aprendizaje, siendo así uno de los modelos más importantes localizado dentro de esta rama.

El paradigma de aprendizaje más comúnmente usado junto a este tipo de modelos es el supervisado. La resolución de tareas como la predicción y clasificación se beneficia mucho de él, aunque también se pueden adaptar a otras tareas típicas de otros modos de aprendizaje.

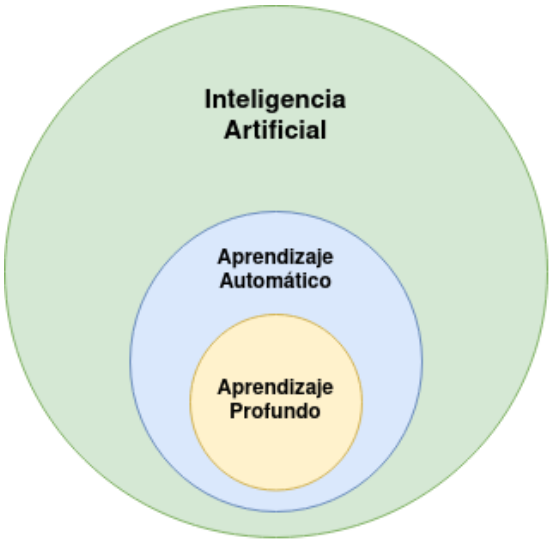

Figura 2.5: Relación entre AI, ML y DL. Fuente: Elaboración propia.

Breve historia del Aprendizaje profundo

Hoy en día podemos encontrar una gran cantidad de modelos propuestos cuya complejidad requiere el uso de supercomputadores para poder ejecutarse y ser útiles gracias a la gran evolución que han sufrido desde sus inicios. Podemos decir que la base de los modelos neuronales surge en el 1943, cuando McCulloch y Pitts proponen un primer modelo que trata de imitar el comportamiento de una neurona biológica (Beamlab [2017]) denominado Unidad Lógica de Umbral (TLU).

La principal finalidad del uso de TLU fue el modelado de funciones lógicas, donde los pesos y el umbral eran ajustados manualmente para adaptarse a las salidas deseadas, por lo que no se puede considerar como un modelo de aprendizaje supervisado. No es hasta el año 1957 donde se descubre el primer algoritmo basado en este modelo de neurona capaz de aprender sus umbrales y pesos, denominado **perceptrón** (Abdullah [2020]). Este algoritmo no se considera aprendizaje profundo porque no es lo suficientemente complejo, pero su invención desencadenó el resto de avances que dieron lugar a los modelos profundos.

El perceptrón fue propuesto por Frank Rosenblant y es considerado un clasificador lineal binario porque solo emplea una única capa de neuronas para emitir su salida. Además, hace uso de una **regla de aprendizaje** para poder ajustar automáticamente los pesos y umbrales de activación. Su creación surgió principalmente como una herramienta clasificadora para la asistencia a la hora de interpretar imágenes del ejército estadounidense. A pesar de ser considerado un gran avance, no era capaz de adaptarse a problemas complejos.

Posteriormente, lo más usual era proponer nuevos modelos basados en el perceptrón para intentar superar su rendimiento base. ADALINE (Insente [2020]) seguía siendo un modelo de una sola capa, pero sustituyó la regla de aprendizaje por una nueva: la regla delta. Bernard Widrow y su alumno Ted Hoff lograron integrar el descenso de gradiente como una de estas reglas.

En el año 1986, David Rumelhart redescubre un algoritmo de entrenamiento compatible con redes multicapa, inspirado en el descenso de gradiente (Rumelhart et al. [1986]). Este algoritmo se denomina *backpropagation* (paso hacia atrás) y permite ajustar los pesos de cada neurona a partir del error cometido por el modelo, propagándolo hacia atrás.

No es hasta mediados de los años 80 dónde se vuelve a generar interés por el DL tras el impacto que tuvo el uso de *backpropagation* para el entrenamiento del perceptrón multicapa. El desarrollo de nuevos *frameworks* para facilitar la construcción de modelos profundos, la rápida evolución del *hardware* y los grandes desembolsos en I+D por parte de empresas como Google o Microsoft han permitido descubrir una gran cantidad de modelos diseñados para solucionar diferentes tareas relevantes en nuestro día a día.

Detallaremos en los siguientes puntos cuestiones específicas relacionadas con la arquitectura de los modelos neuronales actuales, destacando los componentes de la neurona artificial actual (inspirada en la TLU) y su algoritmo de aprendizaje.

2.1.5. La neurona artificial

La mayoría de modelos actuales de Aprendizaje Profundo toman su base del modelo de neurona propuesto por McCulloch y Pitts. A continuación presentaremos los componentes de este elemento fundamental, que también puede actuar como un modelo, aunque no se considere profundo, conocido como **perceptrón** (sección 2.1.4).

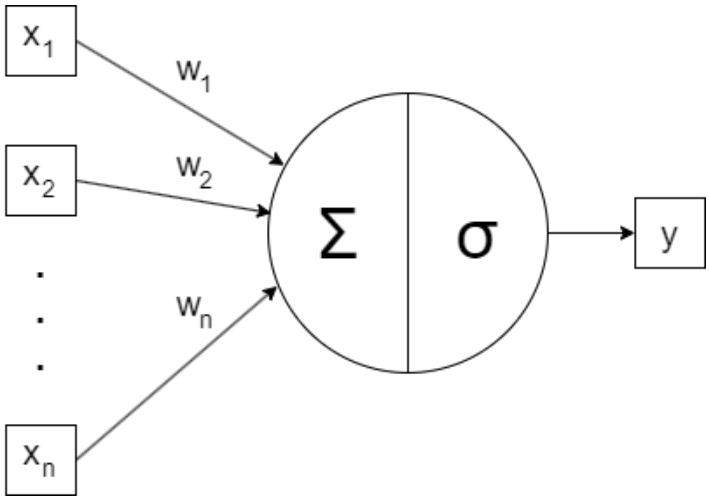

Figura 2.6: Neurona artificial (o perceptrón). Σ representa la combinación lineal de las entradas multiplicadas por sus pesos y σ la función de activación. Fuente: Elaboración propia.

Una neurona (figura 2.6) posee una cantidad específica de entradas ($X = \{x_1, x_2, \ldots, x_n\}$) que son multiplicadas por unos pesos ($W = \{w_1, w_2, \ldots, w_n\}$) y posteriormente agregadas, simulando así el proceso de sinapsis que ocurre entre las neuronas biológicas. Estos pesos permitirán modelar la abstracción de los datos de entrada mediante el uso de funciones lineales (ecuación 2.2).

$$s = \sum_{i=1}^{n} x_i \cdot w_i \tag{2.2}$$

A esta suma ponderada podemos añadirle un sesgo o *bias* (b). Esto permitirá a nuestra neurona contemplar funciones lineales que no siempre pasen por el origen de coordenadas[2] mediante la inclusión de un elemento constante (ecuación 2.3).

$$s = b + \sum_{i=1}^{n} x_i \cdot w_i \tag{2.3}$$

Por último, el valor de salida (y) viene dado por una **función de activación** (σ), la cual puede ser lineal o no lineal (ecuación 2.4). Esta función es la encargada de devolver el valor de activación final y tiene que ser diferenciable. Puede devolver valores discretos (la neurona se activa o desactiva completamente) o continuos (la neurona se activa con cierto grado).

$$y = \sigma(s) = \sigma(b + \sum_{i=1}^{n} x_i \cdot w_i) \tag{2.4}$$

[2]En un espacio de dos dimensiones, la función siempre tomará el valor 0 cuando $X_i = (x_1, x_2) = (0, 0)$ ya que es de la forma $f(X) = WX$. Con el sesgo, tomaría el valor b: $f(X) = WX + b \rightarrow f(X_i) = b$.

Una buena elección de la función de activación puede mejorar drásticamente el rendimiento de nuestro modelo. Originalmente, el perceptrón usaba una función de activación basada en un umbral (θ). La ecuación 2.5 describe su funcionamiento: si la entrada es menor o igual que un umbral, la neurona no se activa y viceversa.

$$\sigma(x) = \begin{cases} 0 \quad si \quad x \leq \theta \\ \\ 1 \quad en \quad otro \quad caso \end{cases} \tag{2.5}$$

La función anterior es lineal, por lo que solo es adecuada para problemas que son linealmente separables, característica que no suelen presentar los conjuntos de datos a día de hoy. Lo anterior pone de manifiesto uno de los problemas presentes en el perceptrón como clasificador binario, tal y como se ilustra en la figura 2.7. Para conseguir fronteras de decisión no lineales, es necesario usar funciones de activación no lineales.

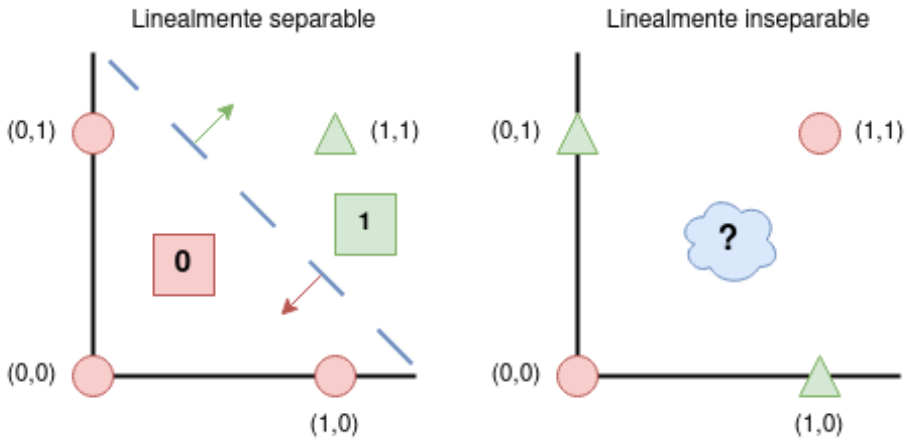

Figura 2.7: Problema linealmente separable y linealmente inseparable. Un perceptrón será capaz de aprender el comportamiento de una puerta AND porque es capaz de modelar una frontera lineal que separe a todas las instancias positivas y negativas a través de la combinación de los pesos aprendidos, datos de entrada y umbrales de activación. Sin embargo, esto no ocurre con la puerta XOR, ya que la separación de instancias no se puede realizar con una función lineal. Fuente: Elaboración propia.

Respecto a la tarea de regresión sucede algo similar, imposibilitando el modelado de funciones complejas que van más allá de lo lineal (por ejemplo: $f(x) = x^2$). La solución a este problema viene dada por el cambio de la función de activación usada, aportando así la no-linealidad necesaria para conseguir que nuestro perceptrón generalice correctamente a partir de los datos proporcionados. Como se puede suponer, la función de activación es un elemento fundamental, importante y que debe de ser adecuada para nuestra tarea. En los siguientes epígrafes describimos algunas (Zhang et al. [2023]).

ReLU (*Rectified Linear Unit*)

$$relu(x) = max(0, x) \tag{2.6}$$

Esta función (ecuación 2.6, figura 2.8) se comporta como la función identidad cuando el valor es positivo, pero devuelve una salida nula en otro caso. El *buen comportamiento* de su derivada es su principal característica diferenciadora, ya que se demostró que mejora el aprendizaje de los modelos que la usan. En 2011, se demostró que mejora aún más el entrenamiento de redes neuronales profundas.

A pesar de no ser diferenciable en todo su dominio debido a la existencia de un punto anguloso en $x = 0$, se toma como derivada en ese punto el valor 0 para solucionar este problema. También hay que tener en cuenta que para entradas negativas el valor de la función toma directamente 0 (y, en su derivada, también) lo que ocasiona la *muerte* de aquellas neuronas que no emitan valores positivos porque no logran aprender.

Por último, indicar que existen una gran cantidad de variantes de esta función de activación: PReLU (He et al. [2015]), Leaky-ReLu o ELU (Bai [2022]) entre otras. Todas intentan mejorar los problemas de ReLU, haciendo especial hincapié en el anteriormente descrito.

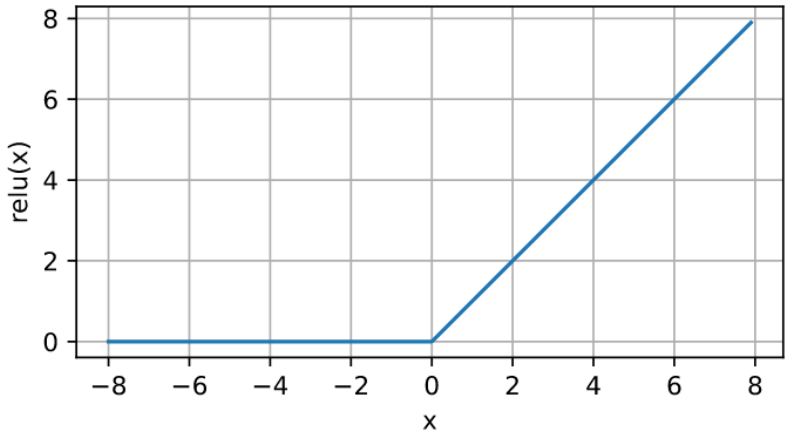

Figura 2.8: Representación de la función de activación ReLU. Fuente: Zhang et al. [2023]

Sigmoide

$$sigmoide(x) = \frac{1}{1 + e^{-x}} \tag{2.7}$$

La función sigmoide (ecuación 2.7, figura 2.9) transforma cualquier valor de entrada al rango $(0, 1)$ y está centrada en $x = 0$ y fue una de las primeras funciones de activación continuas que se propusieron. Se fue sustituyendo por otras (como ReLU) ya que cuando toma valores muy altos o muy bajos su derivada tiende a 0 impidiendo una actualización adecuada de los pesos del modelo a la hora de realizar el paso hacia atrás (*backpropagation*) en modelos con varias capas ocultas.

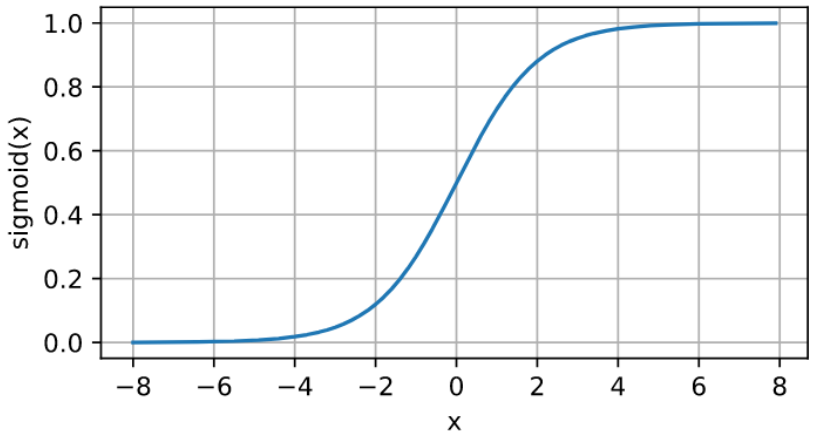

Figura 2.9: Representación de la función de activación Sigmoide. Fuente: Zhang et al. [2023]

Tangente hiperbólica

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{2.8}$$

La tangente hiperbólica (ecuación 2.8) también transforma sus entradas a un rango determinado, en este caso entre $(-1, 1)$. Como se puede apreciar en su representación (figura 2.10) es muy parecida a la sigmoide, por lo que también lo será su derivada y presentando sus mismos problemas.

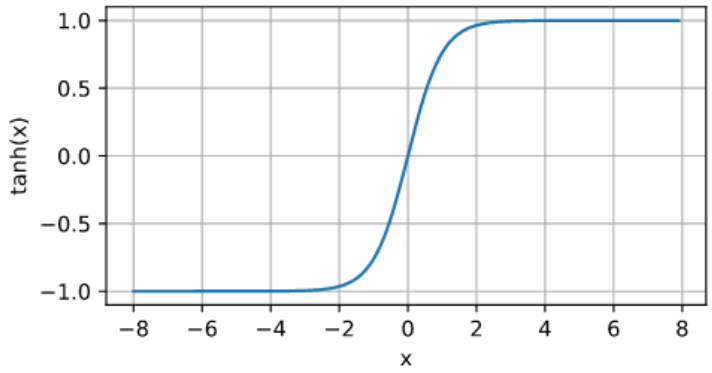

Figura 2.10: Representación de la función de activación tangente hiperbólica. Fuente: Zhang et al. [2023]

Lo habitual es crear modelos con varias capas, donde la salida de las neuronas de una capa sea la entrada de las neuronas de las siguientes. Es muy probable que para conjuntos de datos reales una única neurona no sea suficiente. Lo habitual es crear modelos con varias capas, donde la salida de las neuronas de una capa sea la entrada de las neuronas de la siguiente. A los modelos que siguen la filosofía anterior se les denomina perceptrón multicapa o redes totalmente conectadas (*feed forward network*) y sí tienen la capacidad de considerar profundos. En la siguiente sección comentaremos el funcionamiento general de un modelo multicapa.

2.1.6. Funcionamiento de un modelo neuronal profundo

El uso de una única neurona no es suficiente para hacer frente a los problemas reales en los que se aplica este tipo de algoritmos. Lo más común es crear capas de estas neuronas y apilarlas, de manera que la salida de una capa sea la entrada de la siguiente. Sabemos que este tipo de modelos son supervisados y que aprenden mediante ejemplos, pero todavía no hemos detallado como. En esta sección explicaremos los pasos que realiza una red para emitir su salida y aprender.

En la figura 2.11 se muestra un modelo neuronal profundo con tres tipos de capas diferenciadas por el color que toman sus neuronas. La principal finalidad de este tipo de modelos es aumentar su poder de representación, es decir, incrementar la capacidad que posee para adaptarse a los datos de entrada y desempeñar su tarea correctamente. Una capa de la red es capaz de extraer características relevantes de sus entradas, que a su vez sirven como entrada para la siguiente capa hasta llegar a la capa final. Se dice que las redes neuronales son **estimadores universales** porque

para toda función continua existente podemos encontrar una arquitectura que logre su modelado.

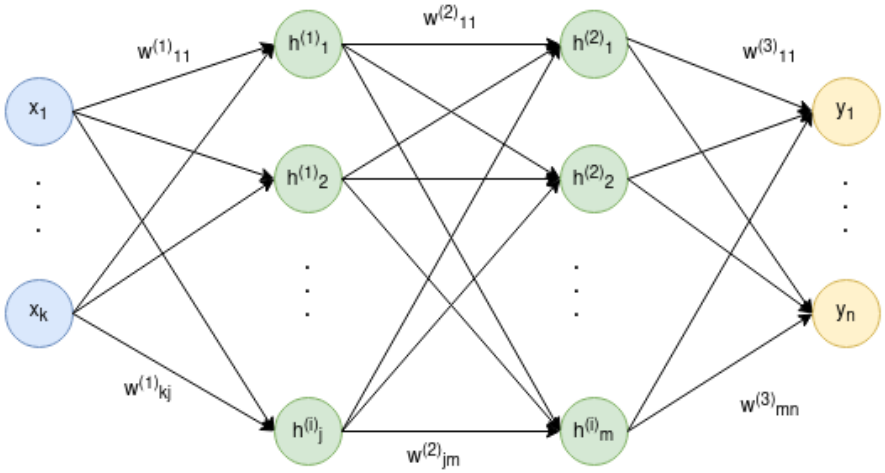

Figura 2.11: Red neuronal profunda. La primera capa se denomina capa de entrada y está representada por el color azul. Las neuronas de esta capa representan las características de entrada y no realizan ninguna transformación sobre ellas. Las capas ocultas están representadas por el color verde, y no tienen por qué tener el mismo número de neuronas que la capa de entrada ni entre ellas. Por último nos queda la capa de salida, representada con el color amarillo. Ella se encarga de calcular el resultado a partir de la salida de la última capa oculta. Fuente: Elaboración propia.

A pesar de que los modelos multicapa son mucho más complejos que una simple neurona, el principio sigue siendo el mismo. Ahora tendremos varios pesos y sesgos por capa, por lo que será necesario identificarlos. En la ecuación 2.9 se muestra como se calcula la salida de una neurona arbitraria localizada en una capa oculta.

$$h_p^i = \sigma(\sum_{l=1}^{k} h_k^{i-1} \cdot w_{k,p}^i + b_p^i) \tag{2.9}$$

Donde:

- h_p^i representa la salida de la neurona p localizada en la capa i
- $w_{k,p}^i$ representa el peso asignado para la entrada k de la neurona p localizada en la capa i.
- b_p^i representa el sesgo de la neurona p localizada en la capa i.
- h_k^{i-1} representa la salida de la neurona k de la capa $i-1$.

Si bien podemos calcular todas las salidas de una capa mediante la expresión anterior, esto no es algo práctico ni recomendable de realizar debido a que no es la manera más óptima de realizar este cálculo. Mediante la vectorización de todo lo anterior podemos

aprovechar mejor la rapidez de las GPU y CPU actuales. Para ello, expresaremos de forma matricial el cálculo de la salida de todas las neuronas de una capa.

La matriz fila $\mathbf{H^i} \in \mathbb{R}^{1\times j}$ contiene las salidas de las j neuronas que posee una capa. Si $i = 0$ contendrá las características originales de la instancia que se va a procesar por la red, por lo que $j = k$.

$$\mathbf{H^i} = \begin{bmatrix} h_1^i & h_2^i & \dots & h_j^i \end{bmatrix} \tag{2.10}$$

La siguiente matriz $\mathbf{W^i} \in \mathbb{R}^{k\times j}$ contiene todos los pesos de una capa de la red, siendo cada fila de la matriz una neurona (j) y cada columna su número de entradas (k).

$$\mathbf{W^i} = \begin{bmatrix} w_{1,1}^i & w_{1,2}^i & w_{1,3}^i & \dots & w_{1,j}^i \\ w_{2,1}^i & w_{2,2}^i & w_{2,3}^i & \dots & w_{2,j}^i \\ \dots & \dots & \dots & \ddots & \dots \\ w_{k,1}^i & w_{k,2}^i & w_{k,3}^i & \dots & w_{k,j}^i \end{bmatrix} \tag{2.11}$$

La matriz fila $\mathbf{B^i} \in \mathbb{R}^{1\times j}$ contiene todos los sesgos de todas las neuronas de una capa.

$$\mathbf{B^i} = \begin{bmatrix} b_1^i & b_2^i & \dots & b_j^i \end{bmatrix} \tag{2.12}$$

A continuación expresaremos la salida de una capa de neuronas arbitraria de forma matricial (ecuación 2.13). Para obtener la salida de la capa i ($\mathbf{H^i} \in \mathbb{R}^{1\times j}$) es necesario usar la salida de la capa anterior ($i-1$) como entradas para multiplicarlas por los pesos correspondientes para cada neurona y entrada, dados por la matriz $\mathbf{W^i}$. Finalmente, sumamos la matriz de sesgos a la matriz dada por $\mathbf{H^{i-1}} \times \mathbf{W^i}$. Por último, aplicamos la función de activación a cada elemento de la matriz anterior para obtener la salida final de la capa.

$$\begin{aligned} \mathbf{H^i} &= \sigma(\mathbf{H^{i-1}} \times \mathbf{W^i} + \mathbf{B^i}) \\ &= \begin{bmatrix} \sigma(\sum_{l=1}^{j} h_l^{i-1} w_{l,1}^i + b_1) & \sigma(\sum_{l=1}^{j} h_l^{i-1} w_{l,2}^i + b_2) & \dots & \sigma(\sum_{l=1}^{j} h_l^{i-1} w_{l,j}^i + b_j) \end{bmatrix} \\ &= \begin{bmatrix} h_1^i & h_2^i & \dots & h_j^i \end{bmatrix} \end{aligned} \tag{2.13}$$

Los cálculos anteriores se realizan para cada capa de la red hasta llegar a la capa de salida, obteniendo así el resultado final. Este paso es el primero que realiza cualquier red neuronal y se le denomina **paso hacia adelante** (*forward pass*). Si un modelo está entrenado, solamente aplicará este paso para obtener la predicción solicitada.

Si queremos entrenar una red neuronal es necesario realizar dos pasos más. El primero de ellos se trata de un cálculo del error a partir del resultado esperado (y) y

del obtenido ($\hat{y}$) que ha cometido el modelo, también llamado *loss* o pérdida. El objetivo principal del algoritmo de aprendizaje será minimizar esta función, por lo que podemos contemplarlo como un problema de optimización.

El segundo paso del entrenamiento realiza esta optimización. Sabiendo que la salida del modelo depende de todos los pesos y sesgos de la red y que, por lo tanto, influyen en el error cometido, nos podemos plantear las dos cuestiones siguientes:

- ¿Cómo varía el error en función de los parámetros de la red?
- ¿Cómo minimizamos el error en función de la variación anterior?

Estas dos cuestiones permiten ver el entrenamiento de la red como un problema de optimización. El algoritmo de *backpropagation* soluciona la casuística planteada en la primera pregunta y el algoritmo de gradiente descendente nos permite realizar esa minimización del error deseada modificando los parámetros del modelo. A este paso se le denomina **paso hacia atrás** porque, como veremos, el ajuste de pesos se realiza de forma iterativa, empezando por la última capa.

La herramienta que nos permitirá saber la variación del error en función de cualquier parámetro es el gradiente, que no es más que una generalización multidimensional de la derivada. Para construirlo deberemos de calcular la derivada de cada peso y sesgo del modelo respecto a la función de error ($\frac{\partial \mathcal{L}}{\partial w_j^i}$ y $\frac{\partial \mathcal{L}}{\partial b_j^i}$) lo cual es computacionalmente costoso y tedioso. *Backpropagation* propone el cálculo de estos gradientes empezando por la última capa hasta llegar a la primera usando la **regla de la cadena**. Primero se calculan las derivadas parciales correspondientes a la última capa, ya que no son tan complejas, y las usaremos para calcular las derivadas de la capa anterior. Este proceso se realiza hasta llegar a la primera capa, obteniendo así el gradiente del error para cada parámetro de la red.

El algoritmo de descenso de gradiente permite optimizar de manera iterativa los parámetros de nuestra red de tal manera que el error sea mínimo. Como su nombre indica, se aprovechará del gradiente, ya que su dirección y magnitud nos guiarán hacia el mínimo buscado. Partiendo de un conjunto de parámetros inicializado aleatoriamente, el algoritmo calculará el gradiente de todos ellos respecto a la función de error para posteriormente modificarlos hasta lograr alcanzar un valor mínimo o cualquier otra condición de parada definida. Esta modificación viene dada por la magnitud del gradiente y puede provocar la no convergencia del algoritmo si es muy grande, por lo que es necesario la inclusión de un nuevo hiperparámetro denominado *tasa de aprendizaje*, encargado de controlar que el cambio sea grande cuando estemos alejados del

mínimo, pero pequeño conforme nos vayamos acercando. Es muy importante controlar el valor de esta tasa debido al papel que ocupa en el proceso de optimización.

En resumen, el algoritmo de aprendizaje de un modelo profundo se divide en dos etapas: el **paso hacia adelante** y el **paso hacia atrás**. En la primera se procesan los datos de entrada para obtener la salida del modelo y en la segunda se ajusta el modelo en función del error entre el resultado emitido y esperado.

2.1.7. Algunos modelos profundos relevantes

Tras conocer el funcionamiento básico de una red pasaremos describiremos un subconjunto de arquitecturas profundas comúnmente usadas junto a sus principales características.

Redes neuronales convolucionales

El principal propósito de este tipo de redes es lograr el procesamiento de imágenes de una forma más eficiente que los perceptrones multicapa, aunque también se pueden usar en otros tipos de datos (Albawi et al. [2017]).

La característica principal de estas redes es el uso de convoluciones, tal y como su nombre indica. Una convolución es un operador que permite aplicar un filtro (denominado *kernel* o núcleo) sobre una imagen *deslizándolo* sobre ella para extraer así sus características interesantes. En la figura 2.12 se muestra cómo se aplica dicho operador para un pixel de una imagen, proceso que se repite hasta recorrerlos todos. Como podemos observar, usualmente este filtró será una matriz cuadrada si la convolución es de 2 dimensiones.

El aprendizaje de este tipo de red neuronal es similar al de un perceptrón multicapa, la única diferencia es que el objetivo de estos modelos es ajustar los pesos contenidos en los filtros descritos para lograr una extracción adecuada de características. Cada neurona poseerá un núcleo de convolución diferente, permitiendo tener mapas de características diversos que facilitarán la resolución de la tarea, los cuales se apilarán y serán procesados por el resto de capas.

Es importante lidiar con el tamaño de estos mapas, que es igual al de la imagen si recorremos todos sus píxeles. Para ello podemos introducir una capa de *pooling* entre las capas convolucionales, encargada de reducir la dimensionalidad de estos mapas,

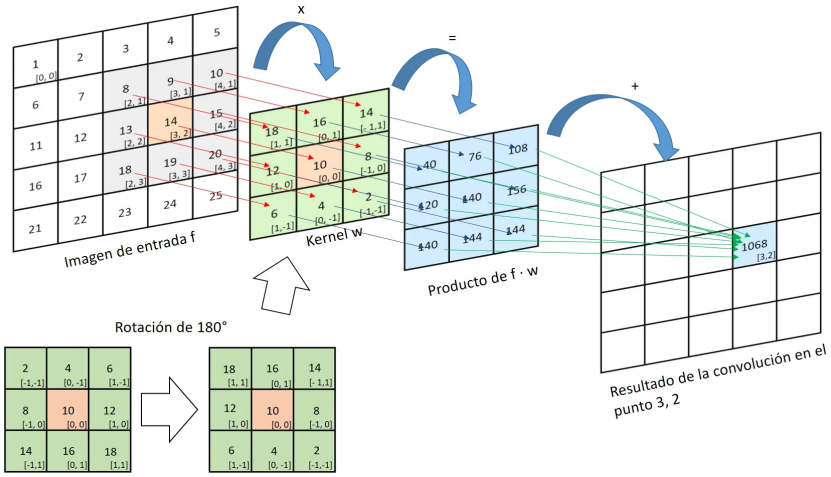

Figura 2.12: Aplicación del operador de convolución sobre una imagen. Fuente: Cuartas [2021]

agregando características de regiones locales. En otras palabras, es muy probable que una característica sea muy parecida a sus vecinas, por lo que es una buena idea agregarlas. En Zafar et al. [2022] se encuentran descritas varias implementaciones de esta capa, como el *max-pooling* (figura 2.13). Su funcionamiento es sencillo, el mapa inicial se divide en regiones iguales y se selecciona el valor máximo de cada región.

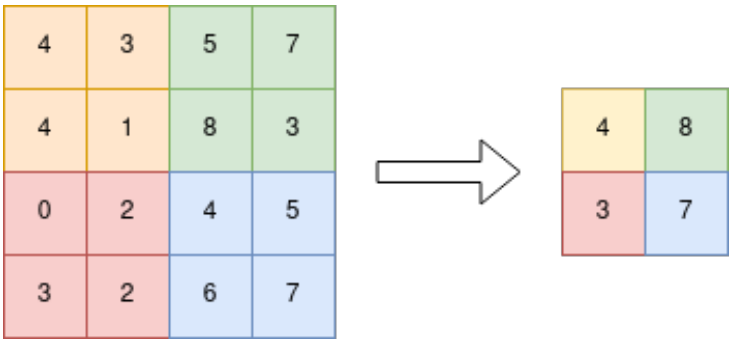

Figura 2.13: Ejemplo de aplicación de *max-pooling* a un mapa de características de tamaño 4x4. Podemos observar que el mapa agregado es de tamaño 2x2. Fuente: Elaboración propia.

Este tipo de redes se pueden aplicar para una gran variedad de tareas. Dependiendo del tamaño y dimensionalidad de la convolución podemos diferenciar entre redes convolucionales 1D, 2D y multidimensionales (3D o más) (Li et al. [2022]).

- Las redes convolucionales 1D son utilizadas para la predicción de series temporales y la identificación de señales.
- Las convoluciones 2D se usan para las tareas de clasificación y segmentación de imágenes, la detección de objetos o el reconocimiento facial.
- Convoluciones de dimensionalidad mayor a las anteriores pueden resolver problemas como el reconocimiento de actividad humana en vídeos o la detección y

reconocimiento de objetos teniendo en cuenta la profundidad (como ocurre en el formato RGBD).

Redes neuronales recurrentes

Las redes recurrentes (RNN) son usadas para procesar datos secuenciales como el texto, las series temporales o incluso genomas (Schmidt [2019]). La principal diferencia entre una RNN y un perceptrón multicapa es cómo se transmite la información por la red. A diferencia del resto de arquitecturas, estas redes presentan ciclos que permiten transmitir información de vuelta a ellas mismas. Dichos ciclos permiten tener en cuenta los diferentes pasos que componen la secuencia de entrada para emitir su salida.

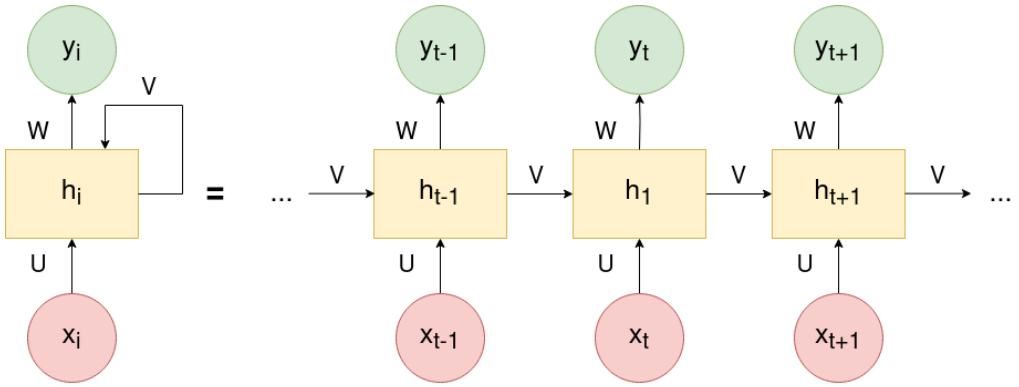

Figura 2.14: Funcionamiento general de una neurona recurrente. U, W y V son pesos, x_i representa un paso de la secuencia de entrada e y_i la salida del modelo para el paso i. Fuente: Elaboración propia.

En la figura 2.14 se ilustra el funcionamiento de una celda recurrente. Para cada elemento de la secuencia de entrada (x_i) se genera un estado oculto (h_i) combinando la entrada con el estado oculto anterior (h_{i-1}), simulando una memoria que agrega información de interés de todos los pasos que se han procesado hasta el momento, que se usará para calcular la salida del modelo y para retroalimentar a la neurona en el procesamiento del siguiente paso. Como también se muestra en la imagen, las neuronas recurrentes poseen pesos de entrada (U), de salida (W) y específicos para la creación del estado oculto (V). Las expresiones 2.14 indican formalmente el funcionamiento general de este tipo de redes.

$$\begin{aligned} h_i &= \sigma(x_i \cdot U + h_{i-1} \cdot V) \\ y_i &= \sigma(h_i \cdot W) \end{aligned} \tag{2.14}$$

Un hiperparámetro relevante en estas redes es la dimensionalidad (o número de características) del estado h. A medida que aumenta se mejorará la capacidad de representación y la memoria del modelo, permitiendo retener durante más tiempo información a corto plazo, además de posibilitar la captación de patrones más complejos. En contraposición, esto conlleva un aumento de la complejidad computacional del modelo y un mayor riesgo de sufrir sobreaprendizaje. A mayor número de características, mayor cantidad de parámetros entrenables y, por consiguiente, mayor cantidad de cálculos a realizar. Por otra parte, si trabajamos con conjuntos de datos pequeños y dotamos al modelo de una memoria más que suficiente para ellos, en su entrenamiento se limitaría a memorizarlos, decayendo así su capacidad de generalización a la hora de abarcar casos desconocidos. En la tabla 2.1 resumimos las conclusiones derivadas de esta discusión.

	Dimensión de h	
	Aumenta	Disminuye
Ventajas	Mayor capacidad de representación	Menor capacidad de representación
	Mayor memoria	Menor memoria
Desventajas	Más riesgo de sufrir sobreaprendizaje	Menor riesgo de sufrir sobreaprendizaje
	Mayor complejidad computacional	Menor complejidad computacional

Tabla. 2.1: Ventajas e inconvenientes de modificar el tamaño del estado oculto (h) de una red recurrente.

Al igual que el resto de arquitecturas, podemos apilar neuronas recurrentes para formar capas que permitan obtener una representación cada vez más abstracta de la secuencia de entrada. La figura 2.15 muestra una arquitectura con dos capas recurrentes junto con las representaciones parciales de la serie. La capa densa emitirá la salida en función de los estados ocultos de dicha representación, que a su vez contendrán información más relevante que la dada por un perceptrón multicapa al ser capaz de tener en cuenta la naturaleza secuencial de las entradas.

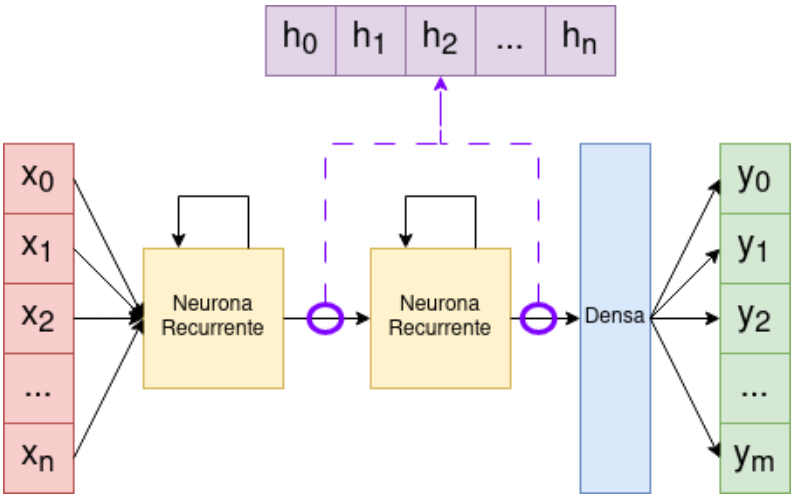

Figura 2.15: Ejemplo de una red neuronal recurrente multicapa. En morado, las representaciones parciales de la serie tras ser procesadas por las neuronas recurrentes. La capa densa la procesará para emitir así la salida de la red. Fuente: Elaboración propia.

Este tipo de redes presentan tres problemas relevantes que pueden influir en el rendimiento del modelo y en su tiempo de entrenamiento e inferencia. La pérdida de dependencias y la explosión o desvanecimiento de gradiente influyen en el rendimiento. Si las series usadas para el entrenamiento son largas, es posible que el estado interno de las celdas recurrentes no sea capaz de mantener las relaciones encontradas en los primeros pasos de la secuencia.

La explosión o desvanecimiento del gradiente también tiene relación con la imposibilidad de captar dichas correspondencias a largo plazo, ya que afecta al aprendizaje de la red. Una red recurrente se puede ver como una red totalmente conectada de varias capas formadas por una única neurona y que emiten una salida para cada paso de la serie de entrada, lo que se conoce como **desenrollado** y se usa junto a una modificación del algoritmo de *backpropagation* clásico, denominado propagación hacia atrás a través del tiempo (BPTT, *BackPropagation Through Time*) para calcular sus gradientes (Lillicrap and Santoro [2019]). Este desenrollado genera redes muy profundas, dando lugar a que el gradiente en las primeras capas sea muy grande (explotando) o muy pequeño (desvaneciendo). Consecuentemente, el ajuste de los pesos de la red puede convertirse en un paseo aleatorio (cambios grandes y bruscos) sin lograr que converjan o que los cambios sean tan pequeños (e incluso nulos) que no permitan a la red aprender (Pascanu et al. [2013]).

Por último, los tiempos de entrenamiento de las RNN pueden llegar a ser altos debido a su imposibilidad de paralelización al depender el estado oculto de cada paso con el del anterior. Para todos los problemas expuestos en estos párrafos existen propues-

tas que intentan solventarlos mediante modificaciones de las neuronas recurrentes o la propuesta de nuevos modelos. Más adelante, se comentarán algunos de ellos.

Modelos generativos

El objetivo principal de los modelos generativos (GM et al. [2020]) es lograr aprender los patrones y distribuciones de un conjunto de datos para generar nuevas instancias inexistentes pero similares a las del conjunto original. Existen una gran cantidad de modelos generativos propuestos que no suelen seguir el paradigma de aprendizaje supervisado (figura 2.16), por consiguiente se dice que la generación suele ser una tarea no supervisada.

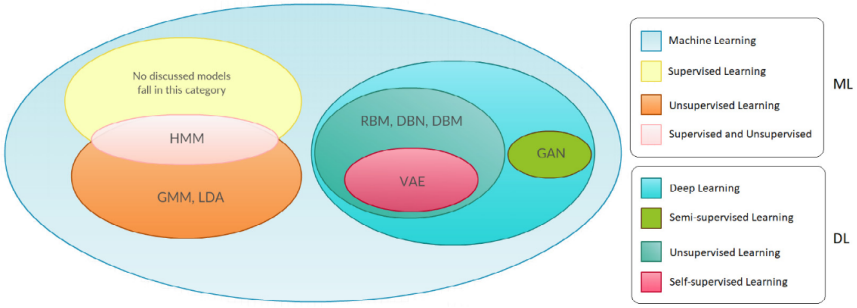

Figura 2.16: Modelos generativos en aprendizaje automático y profundo. Fuente: GM et al. [2020].

Uno de los modelos destacados de este tipo son las redes generativas adversarias (Goodfellow et al. [2014]), formadas por dos submodelos:

- Un modelo **generador** encargado de aprender a crear nuevas instancias sintéticas a partir de los patrones extraídos en su entrenamiento.
- Un modelo **discriminador** encargado de determinar si una instancia proviene del conjunto real de datos. Es un clasificador binario.

El modelo generador tratará de generar instancias sintéticas de cada vez más calidad, hasta que el discriminador logre confundirlas con las reales. Una de las dificultades que presenta este modelo es el entrenamiento de los submodelos, ya que no suele ser a la par y es necesario determinar qué estrategia seguir para entrenar ambos modelos progresivamente.

Además de la generación de instancias sintéticas (lo que se denomina *data augmentation*), los modelos generativos pueden usarse para realizar otras tareas intere-

santes como el escalado, reconstrucción o eliminación de ruido de una imagen (Antoniadis [2023]).

Modelos *Transformer*

Los modelos *Transformer* (Vaswani et al. [2017]) nacen como una alternativa a las redes recurrentes para el procesamiento de secuencia textuales, capaz de procesar la totalidad de sus pasos de manera paralela gracias a un novedoso mecanismo que implementan denominado atención. Además, presentan una arquitectura formada por varios codificadores (*encoders*) y decodificadores (*decoders*) que son usados conjuntamente, pero también independientemente en función de la tarea a resolver.

La arquitectura base de estos modelos se ilustra en 2.17. La secuencia de entrada es procesada en su totalidad por un conjunto de codificadores que implementan un mecanismo de *auto-atención* (*self-attention*) encargada de buscar la similitud entre los diferentes elementos que componen la serie y de crear representaciones parciales de ellos para ser usados posteriormente en los decodificadores, responsables de generar la secuencia de salida de manera autorregresiva. En ellos se implementan otros dos mecanismos de atención denominados *atención enmascarada* (*masked-attention*) y atención cruzada (*cross-attention*). Ambos tienen el mismo funcionamiento que la auto-atención, las principales diferencias son el uso de una máscara para impedir a un paso temporal obtener información de pasos futuros en el primero y el uso de representaciones parciales extraídas del codificador para ver la similitud entre la secuencia generada y la de entrada.

Todas estas atenciones son multicabezal (*multi-headed*). Esto no es más que la aplicación paralela de varios módulos a la vez que luego se agregan para permitir que el modelo extraiga diferentes características de una misma sección de la secuencia de entrada (por ejemplo: una cabeza se podrá centrar en dependencias temporales largas y otra en cortas).

Otro detalle importante tiene relación con la dimensión temporal de los datos. Cómo se procesa toda la secuencia a la vez, el modelo por sí solo no es capaz de identificar qué pasos están antes o después en el tiempo. Es por ello que es necesario añadir información posicional a cada uno para manifestar así ese orden en el tiempo.

Los buenos resultados que ha logrado este modelo en tareas relacionadas con el procesamiento de texto (como la traducción automática o la clasificación de textos) han producido últimamente un gran interés en la comunidad científica. En el tiempo actual

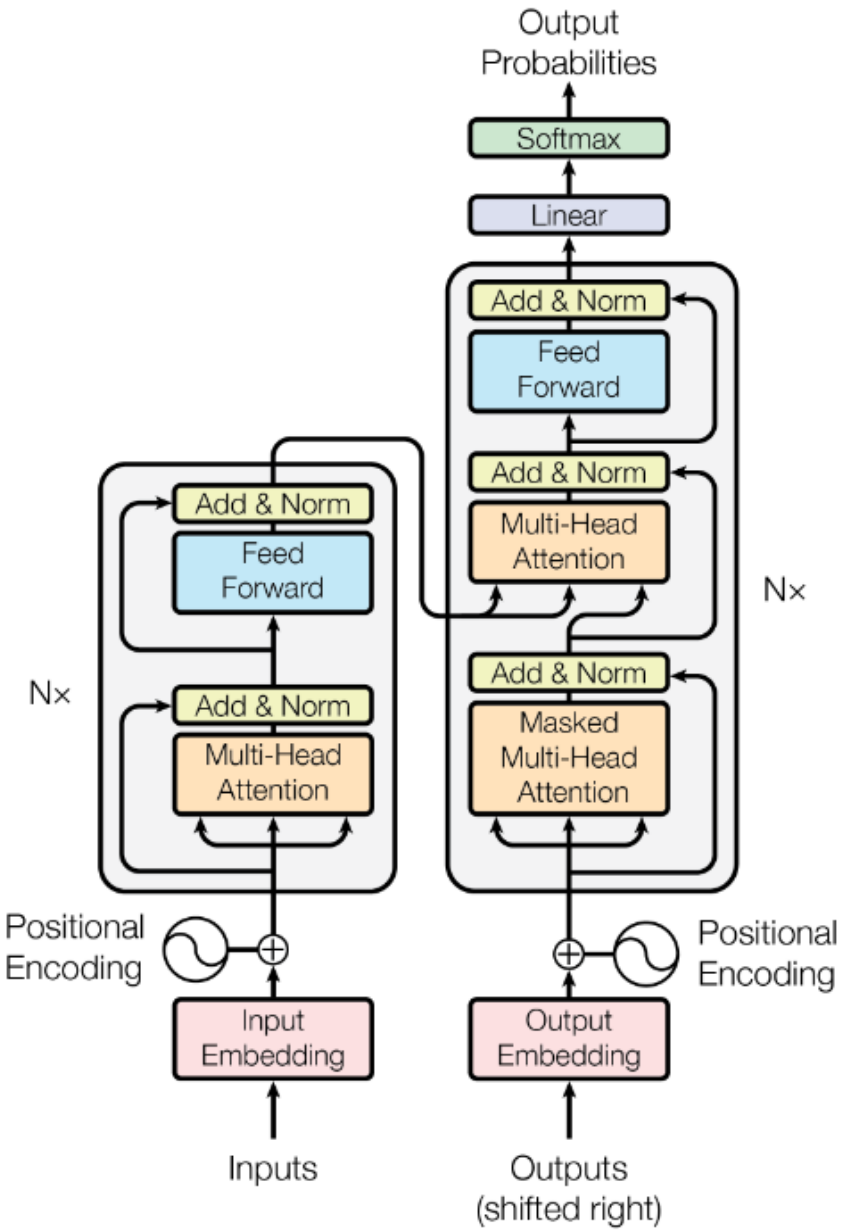

Figura 2.17: Arquitectura básica de un modelo *Transformer*. Fuente: Vaswani et al. [2017]

hay una gran diversidad de literatura respecto a este tipo de modelos y, sobre todo, nuevas propuestas relacionadas con el estudio de mecanismos de atención. Además de adaptar este modelo a nuevas tareas, también se pretende solucionar algunos de sus principales problemas como la gran cantidad de memoria que ocupan y la lentitud del modelo en secuencias largas (Zhou et al. [2021]).

2.2. Series temporales

En esta parte del capítulo describiremos el concepto de serie temporal junto a su tarea de predicción, centrándonos sobre todo en el paradigma del aprendizaje profundo, pero sin dejar a un lado otras metodologías y técnicas relevantes de este ámbito.

Aunque ya se haya introducido el concepto de serie temporal en el primer capítulo, conviene profundizar un poco más sobre él antes de entrar en la tarea de predicción de series temporales. Como se ha mencionado, una serie temporal (ecuación 2.15) no es más que un conjunto de observaciones de un fenómeno (x_n) ordenadas a lo largo del tiempo (n).

$$s_t = \{x_t, t = 1, 2, ..., n\} \tag{2.15}$$

Una de las primeras características a determinar en cuanto queremos analizar una serie es saber si los datos que se recogen son medidos a lo largo de todo el tiempo o en intervalos (regulares o irregulares), siendo así la serie discreta o continua. También es de interés saber si es univariante o multivariante, dependiendo de si en un instante se recogen una o más observaciones.

La naturaleza ordenada de estos datos destaca respecto al resto de tipos que no la poseen. El análisis de series temporales cobra un importante papel a la hora de resolver tareas relacionadas con ellas, siendo además peculiar y diferente porque hay que tener en cuenta su componente temporal. Además de los estadísticos habituales (como la media, varianza, máximos o mínimos) existen otra serie de características propias de este tipo de dato.

2.2.1. Descomposición de una serie temporal

A la hora de afrontar el análisis de una serie temporal conviene descomponerla para poder extraer ciertas características propias que, a simple vista, no son fáciles de identificar. A lo largo del tiempo se han propuesto una gran variedad de descomposiciones, pero la más usada actualmente es la STL (*Seasonal Trend Decomposition based on LOESS*) propuesta en Cleveland et al. [1990].

Este modelo percibe una serie temporal como la agregación ($\circ$) de tres componentes. Dicha agregación puede realizarse de forma aditiva o multiplicativa ($\circ = \{+, \cdot\}$).

$$s_t = t_t \circ e_t \circ r_t \tag{2.16}$$

Donde:

- El componente t_t representa la **tendencia** de la serie temporal, encargada de informar cómo va cambiando la serie a lo largo del tiempo.
- El componente e_t se denomina **estacionalidad** (o componente estacional) e informa de patrones que se repiten con una periodicidad fija y conocida en las mediciones de una serie a lo largo del tiempo. Los patrones estacionales no deben de confundirse con los cíclicos, ya que estos últimos no tienen por qué poseer una frecuencia fija. Normalmente, los patrones estacionales tienen relación directa con el calendario (por ejemplo: cada lunes, cada mes o en Navidad) a diferencia de los cíclicos (por ejemplo: la actividad económica de un país suele presentar ciclos compuestos por una época de auge y otra de decadencia que no presentan siempre la misma duración).
- Por último, el componente r_t representa los **restos** de la serie que no pueden ser explicados mediante la tendencia o la estacionalidad, es decir, el ruido que poseen las mediciones recogidas. Debe de ser lo más pequeño posible para así poder confirmar que el resto de elementos explican correctamente el comportamiento de la serie.

Para decidir si usar una agregación aditiva o multiplicativa nos fijamos en cómo evoluciona la magnitud de los patrones estacionales que posea la serie original. Si son proporcionales al nivel (o valor medio) de la serie, usaremos un modelo multiplicativo. En otro caso, el modelo aditivo será adecuado. En la figura 2.18 se ilustran ambos tipos de estacionalidad.

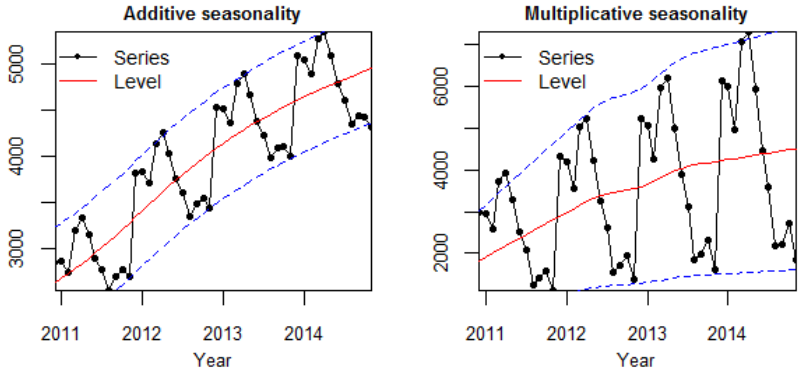

Figura 2.18: Serie temporal con estacionalidad aditiva y multiplicativa. Fuente: Preissler [2018]

2.2.2. Estacionariedad

Ahora describiremos una de las propiedades más relevantes de las series temporales, la **estacionariedad**. Se dice que una serie es estacionaria cuando sus propiedades estadísticas no dependen del momento en el que se observan. Para ello, tenemos que comprobar que para todo desplazamiento temporal de $h \in \mathbb{Z}$ unidades, las propiedades de s_t son similares a las de $s_{t+h} = \{x_{t+h}, t = 1, 2, ..., n\}$. Cabe destacar la diferencia entre estacionariedad **débil** y **fuerte**:

- La **estacionariedad débil** requiere únicamente verificar que la media y la covarianza sean similares para todo desplazamiento h. Siempre que hablemos de esta propiedad, estamos haciendo referencia a esta definición.
- La **estacionariedad fuerte** requiere que **todas** las propiedades estadísticas sean similares para todo desplazamiento h. Consecuentemente, una serie con estacionariedad fuerte también es débil.

Para determinar si una serie es estacionaria se suele usar el test aumentado de Dickey-Fuller o el test aumentado de Engle-Granger. Ambos están descritos en Mackinnon [1990]. Además, existen transformaciones que logran estabilizar las propiedades estadísticas de la serie para conseguir esta propiedad. La más común es restar a cada paso de la serie su valor inmediatamente anterior $(t-1)$ para obtener así la magnitud del **cambio** entre los instantes. A esto se le denomina **diferenciación** y puede ser necesario aplicarla más de una vez.

El resultado inmediato de diferenciar una única vez (primer orden) es la eliminación de tendencias lineales y si volvemos a repetir esta transformación una vez más (segundo orden), eliminaremos aquellas que siguen un orden cuadrático. No es muy

común superar el segundo orden, pero podemos realizar tantas diferenciaciones como sea necesario (orden n).

Podemos también eliminar el componente estacionario mediante el uso de esta técnica. En vez de realizar la diferencia con el instante justamente anterior, usaremos un retraso igual al periodo de nuestra serie. En el caso de que este periodo sea semanal, deberemos restar al valor actual su valor hace 7 días ($t - 7$).

2.3. Predicción de series temporales

Tras conocer más a fondo lo que es una serie temporal junto a sus características y componentes principales, pasaremos a hablar sobre cómo realizar su predicción. El objetivo de esta tarea es estimar los valores que tomará una serie en los próximos T instantes, lo que se conoce como horizonte de predicción, aprovechando las características de la serie. En esta sección nos adentraremos en los diferentes métodos y estrategias que han surgido a lo largo del tiempo para afrontar este problema, aunque nos centraremos sobre todo en cómo se ha ido abarcando desde el punto de vista del aprendizaje profundo.

2.3.1. Estrategias para la predicción de series temporales

En la literatura existen una gran cantidad de propuestas que siguen diferentes estrategias para lograr predecir el comportamiento de una serie a lo largo del tiempo. A continuación hablaremos brevemente sobre cada una, exponiendo sus beneficios y prejuicios (Ben Taieb et al. [2012]):

- **Enfoque autorregresivo o recursivo**: En este enfoque construimos un modelo que es capaz de procesar la secuencia de entrada para devolver su siguiente paso ($\hat{y}_t$). A medida que vamos generando nuevas predicciones, asumimos que son correctas y las añadimos al final de la serie. Esto es, si quiero predecir $\hat{y}_{t+1}$, tomaré como valor verdadero $\hat{y}_t$. Repetimos este proceso hasta que obtengamos el horizonte deseado. A pesar de su sencilla implementación, un problema grave presente en este enfoque es el arrastre del error que se comete porque puede derivar en predicciones muy diferentes a los valores reales.
- **Enfoque directo**: Este enfoque sí permite obtener todo el horizonte de predicción en una iteración a diferencia de la estrategia autorregresiva. Para ello, se

entrena un modelo para cada paso temporal que compondrá este horizonte. Si tenemos una serie cuya frecuencia de muestreo es diaria y queremos predecir un horizonte de longitud 7 (una semana), tendremos que construir 7 modelos (uno para cada día) encargados de estimar cada uno de estos días (el primer modelo para el primer día, el segundo para el siguiente... y así sucesivamente). La única ventaja que nos ofrece es la obtención de todo el horizonte de manera inmediata a cambio de requerir un alto coste computacional al tener que entrenar tantos modelos junto a su incapacidad de percibir dependencias complejas entre los datos al ser construidos independientemente.

- **Enfoque multi-entrada y multi-salida** (MIMO): A diferencia de los otros dos enfoques, el objetivo principal de esta estrategia es usar un único modelo para generar todo el horizonte de predicción a partir de la entrada. De esta manera, podemos tener en cuenta todas las dependencias que se dan entre los instantes de la serie junto a un coste computacional menor al tener que solamente construir un único modelo. La única desventaja es la inflexibilidad que presenta esta estrategia porque los modelos usados en ella no pueden adaptarse fácilmente a cambios en la longitud del horizonte de predicción, siendo generalmente necesario reconstruirlos. Esta estrategia se suele usar junto a modelos provenientes del aprendizaje profundo.

- Enfoques híbridos que combinan las estrategias explicadas anteriormente para obtener lo mejor de cada una.

2.3.2. Modelos clásicos

Las primeras estrategias que se utilizaron para la resolución de esta tarea fueron autorregresivas. Usaban modelos basados en el análisis de la serie para poder modelarla de cara a la obtención de las predicciones, lo que se denomina en la literatura como modelos clásicos o estadísticos. A continuación hablaremos sobre dos algoritmos clásicos relevantes: ARIMA y el suavizado exponencial.

ARIMA

ARIMA proviene de "***A**uto**R**egressive **I**ntegrated **M**oving **A**verage*" que significa "Media Móvil Autorregresiva Integrada" (Fattah et al. [2018]). Este método es la combinación de un modelo autorregresivo (AR) junto a uno de media móvil (MA) y un paso

de diferenciación (I) para lograr la estacionariedad de la serie. La expresión que define a este modelo viene dada en la ecuación 2.17.

$$\hat{y}_t = c + \sum_{i=1}^{p} \phi_i \cdot y_{t-i}^d + \sum_{j=1}^{q} \theta_j \cdot \epsilon_{t-j} \tag{2.17}$$

Observamos que se compone de tres partes, por lo que el analista necesitará especificar tres hiperparámetros para lograr modelar la serie. Los parámetros p e q indican cuántos retrasos (*lags*) debe el componente autorregresivo (primer sumatorio) y de media móvil (segundo sumatorio) usar para emitir la predicción del siguiente paso. El parámetro d hace referencia al número de diferenciaciones que se deben realizar sobre la serie original, por lo que y_{t-i}^d representa el valor de la serie en el instante $t-i$ tras ser diferenciada d veces.

ϵ_{t-j} hace referencia al error de predicción que comete el modelo al estimar el valor del paso temporal $t-j$ y el término c es una constante que suele representar el valor medio de la serie. De ahí que el segundo sumatorio del modelo se conozca como media móvil, ya que si omitimos el componente autorregresivo, las predicciones se construirán incrementando y decrementando dicho promedio.

Existen diferentes modificaciones de este modelo en la literatura, como AutoARIMA y SARIMA (*Seasonal* ARIMA). El primero realiza automáticamente la búsqueda de los parámetros p, q y d mientras que el segundo es capaz de trabajar con series estacionales.

Suavizado exponencial

Esta técnica se basa en la premisa de que la relevancia que toma un paso temporal para predecir el siguiente es inversamente proporcional a su antigüedad. En su versión más simple, realiza una suma ponderada de todos los pasos que componen nuestra serie, asignando a cada paso un peso que se irá reduciendo a lo largo del tiempo (ecuación 2.18). Este método es adecuado para predecir datos sin una tendencia o patrón estacional evidente.

$$\hat{y}_t = \sum_{i=1}^{n} \alpha(1-\alpha)^{(i-1)} \cdot y_{t-i} \tag{2.18}$$

El parámetro α se conoce como factor de suavizado. Este puede ser especificado por el analista, aunque las implementaciones actuales ya cuentan con métodos para encontrar su valor más adecuado.

Este método sirvió de base para el modelo predictor de Holt-Winters (Chatfield [1978]), empleando un enfoque similar al de la descomposición de series. Buscaba modelar el nivel, tendencia y estacionalidad de la serie usando tres suavizados exponenciales y, al combinarlos mediante otra ecuación, poder predecir el siguiente valor. Existen versiones multiplicativas y aditivas de este método, pudiendo así elegir la más adecuada en función de las propiedades de nuestra serie.

2.3.3. Modelos basados en aprendizaje profundo

Hoy en día, la aplicación del aprendizaje profundo sobre esta tarea ha dado lugar a un notable avance. Existen una gran cantidad de propuestas basadas en este paradigma que tratan de explotar la capacidad de las redes neuronales como estimadores universales para identificar los patrones contenidos en una serie y lograr modelarla sin tener que conocer perfectamente sus características, como se hace en los métodos clásicos. En esta sección, detallaremos algunas mejoras de las redes recurrentes y convolucionales (explicadas en la sección 2.1.7) diseñadas específicamente para esta tarea.

Empezaremos describiendo las celdas LSTM y GRU que tratan de resolver el problema de la explosión/desvanecimiento de gradiente dado en las RNN básicas, aclarar que las mejoras añadidas también permitirán a este tipo de redes captar dependencias entre puntos de la secuencia más alejados. Después pasaremos a una propuesta de modelo convolucional, llamado TCN, que en el mejor de los casos permitirá tener en cuenta todo el histórico recogido de la serie para predecir un valor de manera eficiente.

RNN con LSTM (*Long Short Term Memory*) (Hochreiter and Schmidhuber [1997])

Este tipo de celdas (figura 2.19) introducen la creación de tres nuevas puertas (o, como se suele llamar en la literatura, *gates*) junto a la inclusión de un nuevo estado que tomará el rol de memoria a largo plazo de nuestra red (c). Por consiguiente, el estado convencional usado en las RNN básicas (h) representará una memoria a corto plazo.

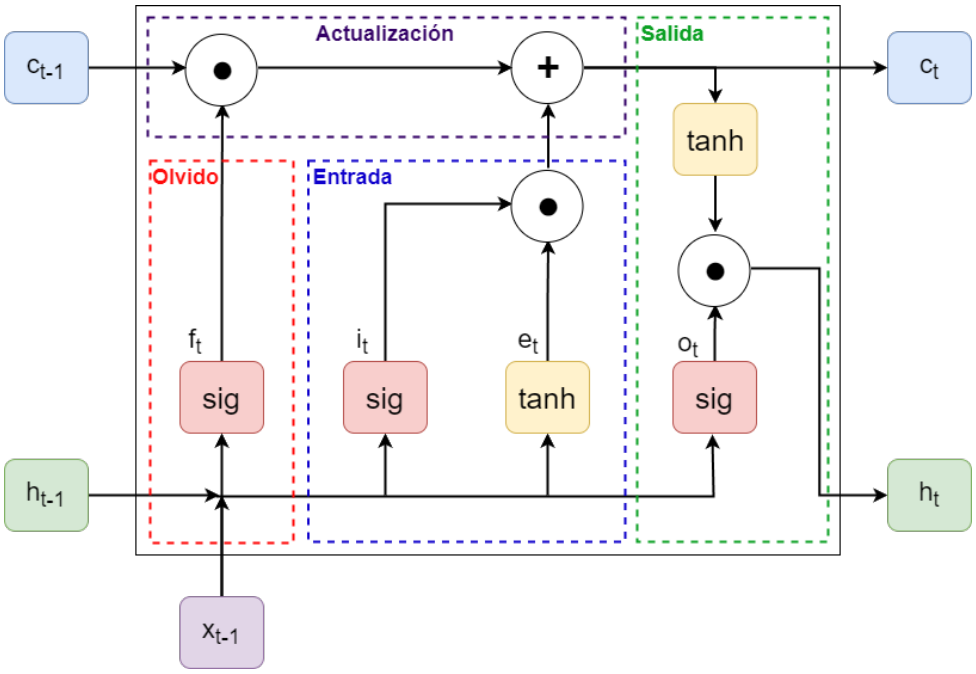

Figura 2.19: Celda LSTM. El cruce entre dos flechas representa una concatenación de lo que representan. El operador • equivale a una multiplicación componente a componente y $+$ una suma. Fuente: Elaboración propia.

El objetivo de estas celdas es aprender a modificar los estados anteriores para controlar el flujo de información a lo largo de la red, determinando cómo se debe actualizar el estado y cuándo debe influir en la salida de la red. Observamos que en las puertas siempre se usarán funciones de activación sigmoidess para forzar que el rango de valores sea $(0, 1)$ para ello. En la notación que sigue, mencionar que la letra W con cualquier subíndice identifican las matrices de pesos que la red aprende durante su entrenamiento.

La puerta de **olvido** (*forget gate*) permite determinar que información deja de ser relevante en función del estado oculto (estado h) y los datos de entrada. Esta puerta se calcula siguiendo la expresión:

$$f_t = \sigma(x_t \cdot W_{tx} + h_{t-1} \cdot W_{th}) \tag{2.19}$$

Esta matriz f_t se multiplica con la memoria a largo plazo c_t para determinar cuánto olvidar en cada paso. Si su valor es cercano a 1, no se olvidará apenas nada y viceversa.

La puerta de **entrada** (*input gate*) controla cuánta información debe de almacenarse en la memoria a largo plazo. Realizamos primero un cálculo similar al de la puerta

de olvido que nos permita cuantificar entre $(0, 1)$ la proporción relevante:

$$i_t = \sigma(x_t \cdot W_{ix} + h_{t-1} \cdot W_{ih}) \tag{2.20}$$

Seguidamente, obtenemos una representación parcial del paso actual junto a la memoria a corto plazo. Le llamaremos nodo de entrada (o *input node*, e_t), y se fusionará con i_t para determinar la nueva información a incluir en la memoria a largo plazo de la celda (c_t):

$$e_t = tanh(x_t \cdot W_{ex} + h_{t-1} \cdot W_{eh}) \tag{2.21}$$

$$c_t = c_{t-1} \cdot f_t + e_t \cdot i_t \tag{2.22}$$

Por último, siguiendo la filosofía del resto de puertas, en la puerta de **salida** queremos determinar si la información contenida en la memoria a largo plazo de la célula deben de influenciar en la salida del paso actual, calculando o_t para combinarlo con c_t justo como se especifica en la ecuación.

$$o_t = \sigma(x_t \cdot W_{ox} + h_{t-1} \cdot W_{oh}) \tag{2.23}$$

$$h_t = tanh(c_t) \cdot o_t \tag{2.24}$$

El uso de estas dos memorias y puertas permite reducir la probabilidad de sufrir explosión o desvanecimiento del gradiente a la hora de aplicar *backpropagation*, permitiendo también a la red apreciar dependencias temporales más duraderas. Podemos calcular la predicción $\widehat{y}_{t+1}$ a partir del último estado h_t, el cual contendrá la información relevante para el paso temporal x_t junto a la información seleccionada de la memoria a largo plazo (c_t).

RNN con GRU (*Gated Recurrent Unit*) (Chung et al. [2014])

Siguiendo la filosofía de diseño basada en puertas de las celdas LSTM surge la celda GRU. Se caracteriza por usar únicamente un estado oculto (h) y dos puertas denominadas como **reinicio** (*reset*) y **actualización** (*update*). Al igual que en el modelo anterior, controlarán qué información se elimina o incluyen en dicho estado. En el esquema de la figura 2.20 se ilustra el funcionamiento de este tipo de celda.

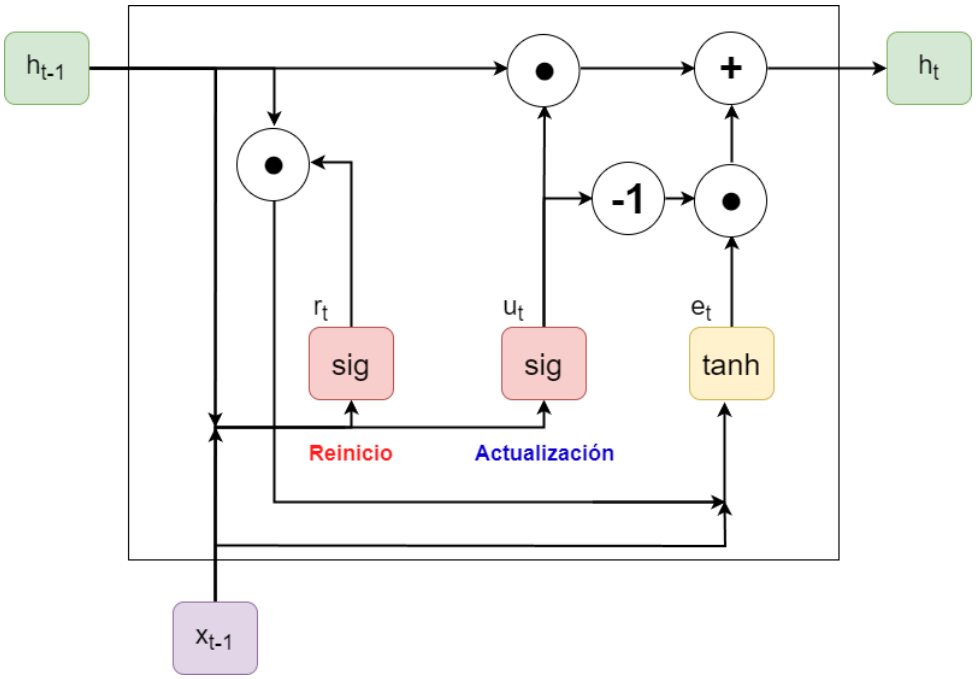

Figura 2.20: Celda GRU. El cruce entre dos flechas representa una concatenación de lo que representan. El operador • equivale a una multiplicación componente a componente y + una suma. Fuente: Elaboración propia.

La puerta de **reinicio** (*reset*) permite determinar qué es necesario olvidar. Se calcula mediante una activación sigmoide siguiendo la ecuación 2.25.

$$r_t = \sigma(h_{t-1} * W_{hr} + x_t * W_{xr}) \tag{2.25}$$

Como se puede intuir, una vez que hemos determinado la proporción de información a olvidar, modificamos el estado oculto de la celda. Al igual que con las puertas de la celda LSTM, cuanto más cercano a 1 sea el valor de esta puerta, menos modifica el estado y viceversa. Obtenemos un estado oculto parcial h^r_{t-1} a partir de la operación 2.26.

$$h^r_{t-1} = h_{t-1} \cdot r_t \tag{2.26}$$

La puerta de **actualización** (*update*) determina qué información del estado oculto anterior debe de mantenerse en el futuro. Se calcula tal y como se describe en la ecuación 2.27.

$$u_t = \sigma(h_{t-1} * W_{hz} + x_t * W_{xz}) \tag{2.27}$$

Generamos el estado candidato a partir de una activación hipertangente, usando una concatenación del estado oculto parcial h_{t-1}^r y la entrada actual con sus respectivos pesos.

$$e_t = tanh(h_{t-1}^r * W_{hh} + x_t * W_{xh}) \tag{2.28}$$

Y por último, generamos el estado oculto actual mediante la combinación del estado oculto candidato con el estado oculto anterior en la ecuación 2.29. Aclarar que si u_t indica qué información del estado oculto anterior debe de continuar hacia adelante, $(1 - u_t)$ representará qué información del estado candidato debe de seguir. A medida que el valor de u_t se acerca a 1, el valor de h_t se asemejará más al de h_{t-1} indicando que el instante actual no es tan relevante para realizar la predicción del siguiente.

$$h_t = (1 - u_t) \cdot e_t + h_{t-1} \cdot u_t \tag{2.29}$$

La incorporación de las puertas descritas estabiliza el aprendizaje e impide la explosión o desvanecimiento del gradiente, pero utilizando una menor cantidad de parámetros. Esto conlleva un entrenamiento más sencillo junto a un aumento de la eficiencia, pudiendo tener incluso un rendimiento mayor respecto a las LSTM.

Redes convoluciones temporales

Este tipo de redes aparecen por primera vez en Bai et al. [2018], donde se afirma que son capaces de superar el rendimiento de las redes recurrentes en un amplio número de tareas relacionadas con el modelado de secuencias porque “combinan algunas de las mejores prácticas de las arquitecturas convolucionales modernas”. A continuación detallaremos estas mejoras progresivamente, hasta llegar a la arquitectura final.

Aunque las reden convolucionales convencionales con núcleo unidimensional pueden usarse para predecir el siguiente paso en una secuencia, no son del todo adecuadas. Al procesar un paso temporal t, tienen también en cuenta instantes futuros a él ($t + 1$, $t + 2$...) lo cual no es adecuado al no respetar el orden temporal de los datos.

Es por ello que surgen las convoluciones causales. Ellas corrigen este problema mediante la aplicación de un desplazamiento (*padding*) de la secuencia en función del tamaño de su núcleo, solucionando el problema anterior. En la figura 2.21 se ilustra la diferencia entre una convolución convencional y una causal.

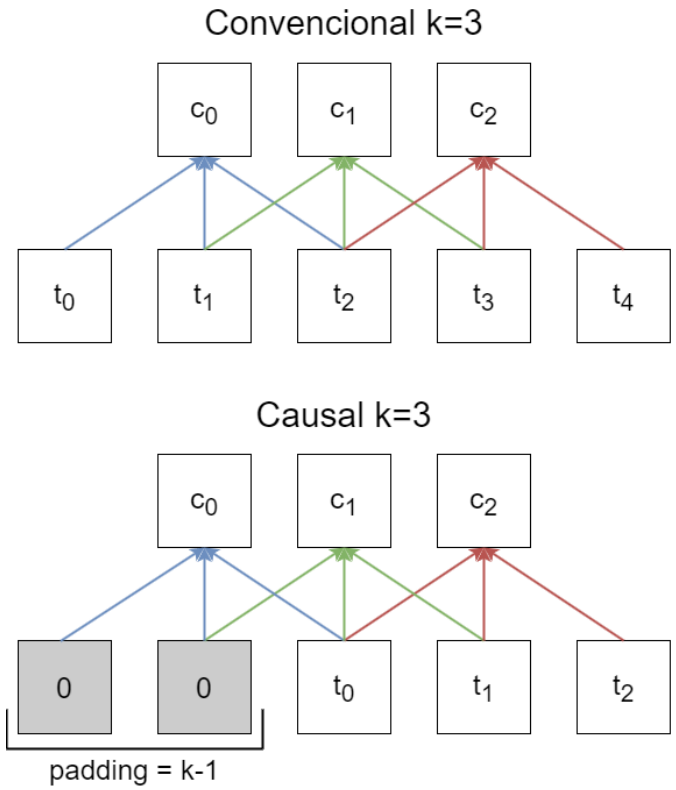

Figura 2.21: Convolución convencional y causal con núcleo de tamaño $k = 3$. Los índices $t_n; n \in \mathbb{Z}$ representa un paso de la serie y $c_m; m \in \mathbb{Z}$ pasos de la convolución. Como se puede apreciar, las convoluciones causales respetan el orden temporal al procesar únicamente todo instante n menor o igual que m gracias al *padding*. Fuente: Elaboración propia

Otras ventajas del uso de este tipo de convoluciones son la capacidad de paralelizar el procesamiento de los pasos temporales y la ausencia de conexiones recurrentes. Esto deriva generalmente en una reducción del tiempo de entrenamiento, haciendo especial hincapié en ello al procesar conjuntos de datos con series muy largas según van den Oord et al. [2016]. Además, también preservan la dimensionalidad de los datos a la hora de emitir su salida. Si introducimos una serie temporal de tamaño m, este se preserva tras aplicar este operador sin necesidad de aplicar técnicas de *pooling* (figura 2.12).

En este mismo artículo también se nos presenta el principal inconveniente del uso de estas convoluciones. Para lograr que el conjunto de pasos que influyen en la predicción sea considerable, es necesario apilar una gran cantidad de capas. A este número de pasos se le denomina **campo receptivo** (o recubrimiento) de la red y, como se observa en la ilustración 2.22, la relación entre número de capas y magnitud de él no es muy óptima usando convoluciones causales.

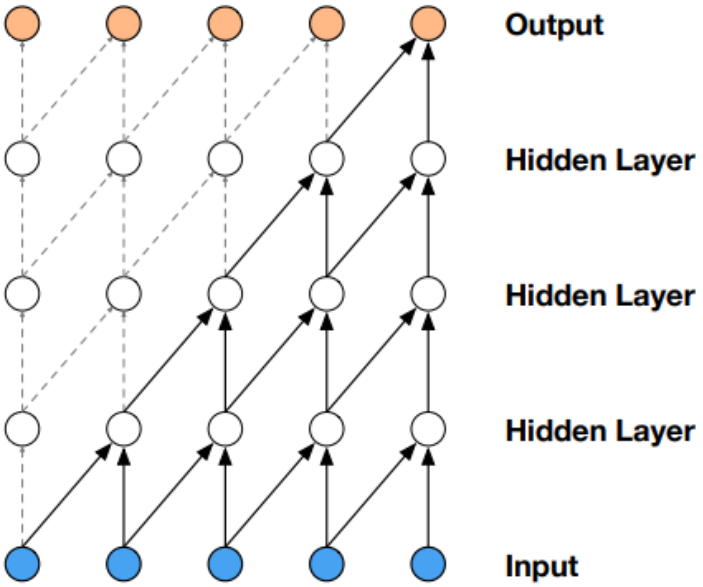

Figura 2.22: Red convolucional causal con tamaño de núcleo $k = 2$. Podemos ver que el campo receptivo equivale a 5 instantes de la serie de entrada y que, para ello, es necesario introducir 3 capas ocultas. Fuente: van den Oord et al. [2016].

Para aumentar el número de entradas que están involucradas en el campo receptivo de la red sin necesidad de incrementar el número de parámetros de ella, se introdujo una nueva mejora del operador de convolución causal, mediante la aplicación de un factor de **dilatación**.

Una convolución dilatada introducen huecos en su filtro para lograr aumentar la información que percibe. En otras palabras, si una convolución tiene factor de dilatación d, tomará los instantes de entrada aplicando una separación de esa misma cantidad de pasos a partir del que se está procesando actualmente.

En la figura 2.23 se muestra una red neuronal convolucional causal y dilatada con 5 capas. Podemos ver cómo el uso del factor de dilatación permite aumentar exponencialmente la magnitud del campo receptivo, aplicando en cada capa una separación cada vez más grande. Gracias a ello, el modelo percibe dependencias temporales más largas sin aumentar su coste computacional y si aumentar el número de parámetros entrenables.

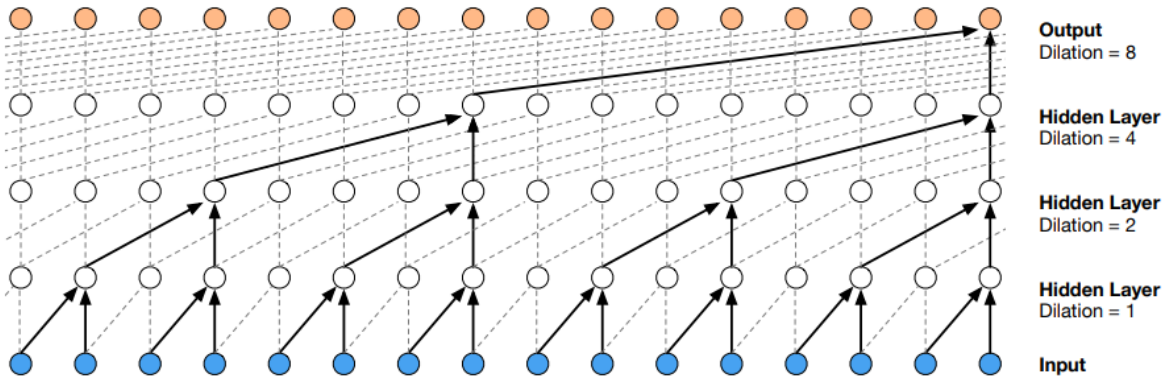

Figura 2.23: Red neuronal convolucional causal y dilatada. Se usa un factor de dilatación del orden de $d = 2^i$, donde i es el número de capas. La capa de entrada no posee factor de dilatación, por lo que la numeración comienza con la primera capa oculta, siendo esta la número 0 ($d = 2^0 = 1$). Observamos que una convolución causal dilatada con $d = 1$ es equivalente a una convolución causal sin dilatación. Fuente: van den Oord et al. [2016]

A pesar de lograr mejorar el campo receptivo de nuestra red, puede seguir siendo necesario apilar una gran cantidad de capas si queremos alcanzar un recubrimiento máximo, dando lugar a redes densas y profundas. La última mejora busca estabilizar las redes TCN apilando **bloques residuales** en vez de capas convolucionales.

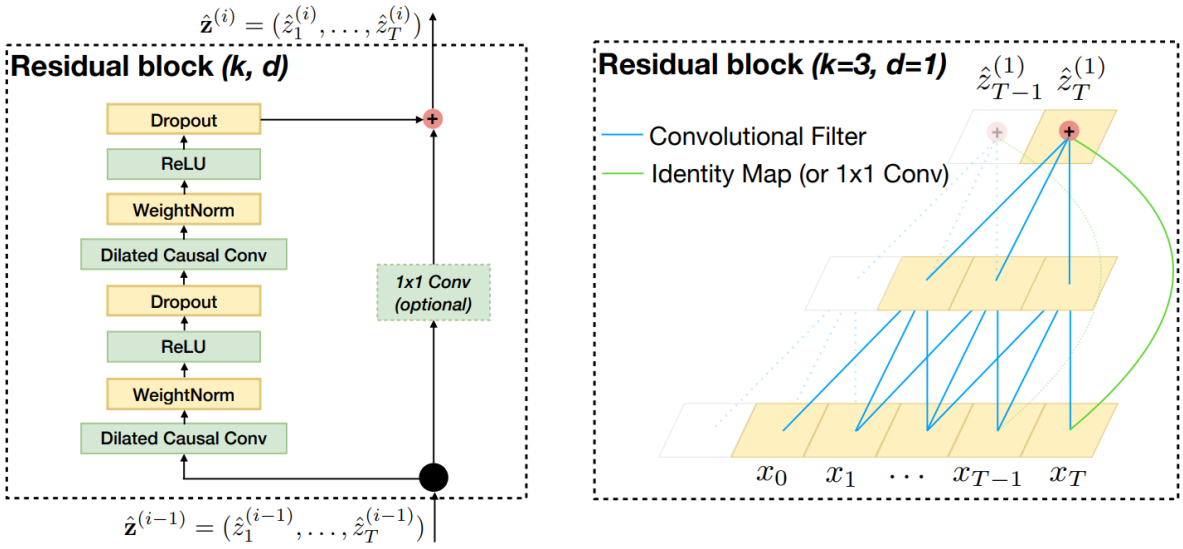

Figura 2.24: Bloque residual de una TCN. k indica el tamaño del núcleo y d la dilatación de las convoluciones causales dilatadas usadas en él. Puede ser necesario aplicar una convolución 1x1 para permitir la agregación del último paso de la entrada con el de la salida de la segunda convolución, ya su número de canales puede diferir. Fuente: Bai et al. [2018]

Un bloque residual (figura 2.24) se compone de dos capas formadas por una convolución causal dilatada junto a la aplicación de una función de activación Re-Lu a su salida para dotar al modelo de no-linealidad. Además, en el entrenamiento se realiza

una normalización de los pesos contenidos en la convolución para acelerar el entrenamiento (Salimans and Kingma [2016]) junto a la aplicación de un *dropout* espacial para evitar el sobreaprendizaje. Como paso final, se emplea una conexión residual para añadir a la salida del bloque el valor del último paso temporal.

La combinación de todas estas mejoras dota de paralelismo, gradientes estables y un campo receptivo flexible a este modelo, así como un bajo uso de memoria en el entrenamiento respecto a las redes recurrentes y permitiendo procesar series de longitud variables.

Capítulo 3

MATERIALES Y MÉTODOS

Tras haber estudiado el contexto que engloba a nuestra tarea, así como el funcionamiento de sus modelos más relevantes, pasamos a describir todos los materiales y métodos que se usarán posteriormente en la experimentación a realizar. Hablaremos sobre los conjuntos de datos usados, su preprocesamiento, los modelos que se han implementado y el marco experimental diseñado para este trabajo.

3.1. Selección de herramientas de desarrollo

Empezaremos este capítulo explorando los diferentes lenguajes de programación usados en el campo de la Inteligencia Artificial y de la Ciencia de Datos. Es importante tener en cuenta criterios como la experiencia personal, popularidad y, sobre todo, su capacidad para desarrollar modelos de aprendizaje profundo. Tras seleccionar el lenguaje, determinaremos que librerías a usar para la implementación de nuestros modelos.

3.1.1. Lenguajes de programación

Existen una amplia variedad de lenguajes de programación diseñados para cumplir diferentes propósitos. En el campo de la Ciencia de Datos podemos destacar tres lenguajes que han acaparado la atención de sus programadores, aunque no sean los únicos. *Python*, *R* y *Java* encabezan este ranking (Luna [2023]) debido a su popularidad y a la gran cantidad de herramientas que poseen para el análisis y tratamiento de datos, además de la construcción de modelos a partir de ellos. El índice PYPL (*PopularitY of Programming Language Index*) corrobora el renombre de estos lenguajes, ocupando el primer, segundo y sexto puesto respectivamente en su ranking de popularidad (tabla 3.1)

Ranking	Lenguaje	Popularidad
1	**Python**	**28.20 %**
2	**Java**	**15.73 %**
3	JavaScript	8.91 %
4	C/C++	6.80 %
5	C#	6.67 %
6	**R**	**4.59 %**
7	PHP	4.54 %
8	TypeScript	2.92 %
9	Swift	2.77 %
10	Objetive-C	2.34 %

Tabla. 3.1: Los 10 lenguajes más populares según el índice PYPL en 2023. Fuente: PYPL [2023].

La gran ventaja que presenta *Java* respecto a *R* y *Python* es su independencia respecto a la plataforma en la que se ejecuta gracias al uso de una máquina virtual. Sin embargo, no presenta tantas herramientas como los otros dos lenguajes para nuestro propósito. Debido a ello, podemos decir que *R* es una buena alternativa a *Java* porque ha sido diseñado específicamente para el análisis estadístico y representación visual

de información, por lo que contiene una gran cantidad de paquetes relacionados con el aprendizaje automático debido a su propósito. A pesar de que *R* es una buena opción, se ha decidido usar *Python* como lenguaje debido a que contiene gran cantidad de las funcionalidades ofrecidas en *R* gracias a su comunidad (siendo algunas de ellas las más usadas en el mundo de la Ciencia de Datos), presenta una sintaxis más sencilla que *R* y la experiencia personal del autor de este trabajo en él es mayor.

3.1.2. Librerías y marcos de desarrollo para aprendizaje profundo

Existe una gran cantidad de herramientas que nos permiten el desarrollo de nuestros modelos en *Python*. No obstante, aunque parezca muy similares, cada una fue producida para cubrir ciertas necesidades específicas (ya sean de un desarrollador independiente o de una gran empresa). Por lo tanto, entre tantas herramientas, es necesario filtrar y seleccionar aquellas que cumplan los siguientes requisitos:

- Que sus creadores sigan ofreciendo **mantenimiento** a ella mediante la presencia de una buena documentación y su actualización constante. A ser posible, que sea también un proyecto ***open-source*** para poder acceder a su código en cualquier momento y poder ser mejorado por la comunidad.
- Que soporte el **uso de GPU** para acelerar los cálculos realizados durante el entrenamiento e inferencia del modelo. Concretamente, necesitaremos que sea compatible con CUDA.
- Que su uso sea **gratuito**, evitando así un gasto que puede llegar a ser considerable, viendo el precio de otras herramientas usadas en otros campos (como la edición de contenido audiovisual o la creación de videojuegos).

En la tabla 3.2 se presentan varios *frameworks* que podemos usar para cumplir nuestros objetivos. Las columnas de la tabla nos informan de la longevidad de la herramienta, si se sigue manteniendo de alguna forma, lenguajes compatibles con ella y si soporta CUDA. Destacar que todas ellas son de código abierto.

Software	Año de salida	En mantenimiento	Lenguaje	Soporta CUDA
Theano	2007	No	Python	Si
Caffe	2013	No	C++	Si
Intel Data Analytics Acceleration Library	2015	N/A	C++, Python, Java	No
Apache SINGA	2015	N/A	C++	Si
Chainer	2015	No	Python	Si
Apache MXNet	2015	No*	Small C++ core library	Si
TensorFlow	2015	Si	C++, Python, CUDA	Si
BigDL	2016	N/A	Scala	No
Microsoft Cognitive Toolkit (CNTK)	2016	No	C++	Si
PyTorch	2016	Si	Python, C, C++, CUDA	Si
PlaidML	2017	Si	Python, C++, OpenCL	No

Tabla. 3.2: Librerías y frameworks de código abierto para aprendizaje profundo en Python. Aunque haya herramientas que no presenten a nuestro lenguaje seleccionado como lenguaje, existen interfaces que permiten su uso desde él. Fuente: Adaptado de Yalcin [2021]. * A diferencia de lo que declara la fuente original, *Apache MXNet* ha sido designado como ”proyecto retirado“ por sus desarrolladores en septiembre del año 2023 (Apache Software Foundation [2024]).

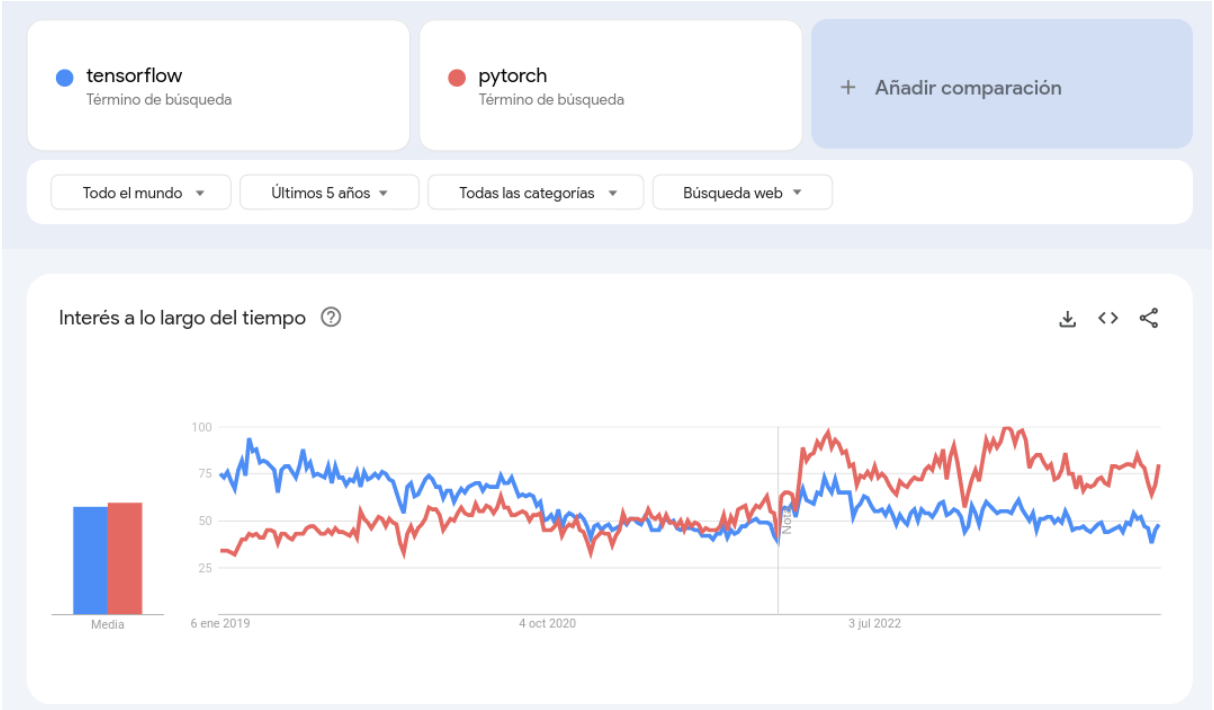

Figura 3.1: Interés en los últimos 5 años de los términos ”tensorflow“ y ”pytorch“. Fuente: Trends [2024]

Tras la revisión realizada, acotamos nuestra búsqueda a únicamente dos *frameworks*, *TensorFlow* y *PyTorch*, ya que son los únicos que poseen un mantenimiento activo, además de ser compatibles con CUDA. La documentación, recursos y comunidad de ambos son excelentes, por lo que cualquiera sería apto para nuestra tarea. Sin embargo, en este proyecto se ha tomado la decisión de usar ***PyTorch*** porque presenta un mayor interés en la comunidad, tal y como reflejan las figuras 3.1 y 3.2. El interés por él y la [1]cantidad de artículos científicos que lo usan justifican su relevancia actual. A pesar de su popularidad, los desarrollos en [2]*Tensorflow* suelen ser más sencillos, ya que no requiere detallar algunos aspectos como los bucles de entrenamiento y test.

[1]Artículos científicos *open-access* cuyo código está disponible en GitHub (Papers With Code [2024]).
[2]En su versión 1.

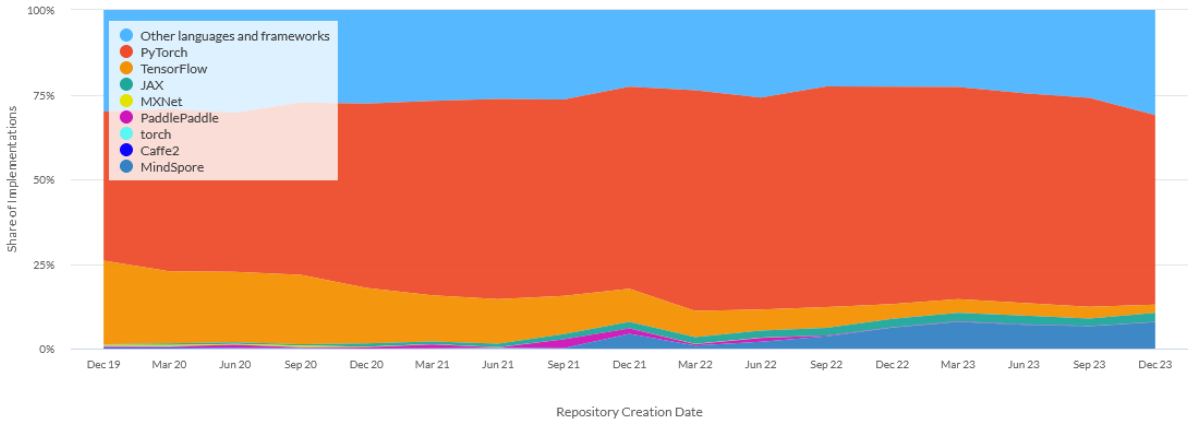

Figura 3.2: Porcentaje de implementaciones por *framework* registrados en la web PapersWithCode desde diciembre del 2019 hasta diciembre del 2023. Fuente: Papers With Code [2024]

Para dotar a *PyTorch* de esa interfaz a alto nivel que no tiene usaremos la librería *PyTorch Lightning*. Gracias a ella dispondremos de diferentes clases que nos ayudarán a implementar los modelos sin necesidad de preocuparse por la implementación de aspectos como los bucles mencionados, el paso de tensores a dispositivos gráficos o el uso de estrategias para realizar un entrenamiento multi-GPU.

3.2. Conjuntos de datos seleccionados

Tras una exploración por diferentes portales para determinar qué conjuntos de datos a utilizar, se ha optado por la utilización del **repositorio para la predicción de series temporales de la Universidad de Monash** (Godahewa et al. [2021]). Su misión es ser el "primer repositorio exhaustivo de previsiones de series temporales que contiene conjuntos de datos de series temporales relacionadas para facilitar la evaluación de modelos de previsión globales" gracias a sus 30 conjuntos heterogéneos, con propiedades diferentes y provenientes de diferentes dominios. Resaltar que existen diferentes versiones de cada uno, subiendo el total de conjuntos a 58.

En la sección 3.2.1 detallaremos qué conjuntos se han escogido, su dominio y realizaremos un breve análisis exploratorio de cada uno, que nos será útil para determinar el preprocesamiento descrito en la sección 3.2.2

3.2.1. Descripción de los conjuntos de datos

De todos los conjuntos disponibles, hemos filtrado todos aquellos en los que no se especificaba claramente el horizonte de predicción en el repositorio y no eran univariables. Posteriormente, nos hemos quedado únicamente con los siguientes 8 conjuntos principales (ya que se han seleccionado varias versiones de algunos):

- **Electricity**: Contiene el consumo eléctrico de 370 clientes en kilovatios desde el 2011 hasta el 2014. Todas las series poseen la misma longitud. Solo se ha seleccionado la versión cuya frecuencia de muestreo es semanal.
- **M1**: Este conjunto de datos se compone de una selección de series de una de las primeras competiciones dedicadas a su predicción, denominadas competiciones M (M, porque su creador se llama Makridakis). Consta con 617 series de tiempo mensuales, relacionadas con la economía, la industria y la demografía. No todas las series presentan el mismo número de mediciones. De él hemos seleccionado las versiones que presentan una frecuencia de muestreo mensual y cuatrimestral.
- **M3**: Como el conjunto de datos M1, contiene 1428 series de la tercera edición de las competiciones M. Vuelven a no tener la misma longitud y añade a sus dominios fenómenos relacionados con las finanzas. De él hemos seleccionado las versiones que presentan una frecuencia de muestreo mensual y cuatrimestral.
- **NN5**: Conjunto de datos de otra competición, con 111 series que recogen transacciones de cajeros automáticos localizados en Gran Bretaña. En este caso, las series presentan una misma longitud. De él hemos seleccionado las versiones que presentan una frecuencia de muestreo diaria y semanal.
- **San Francisco traffic**: Contiene 862 series que recogen las tasas de ocupación de las autopistas localizadas en la Bahía de San Francisco entre los años 2015 y 2016. Solo se ha seleccionado la versión cuya frecuencia de muestreo es semanal.

Gracias a la información proporcionada por el portal del repositorio y al uso de un formato muy similar a ARFF (`.ts`), se ha conseguido extraer fácilmente las características principales de todos los conjuntos. Además, también se cuenta con un *script* desarrollado por los creadores para procesar estos archivos y obtener la información directamente (Godahewa [2021]). En la tabla 3.3 se muestra la información esencial de cada conjunto de datos. Podemos ver que ningún dataset presenta valores perdidos, y que todos contienen una gran cantidad de series (aunque en algunos casos no

posean la misma longitud). También se informa del horizonte que se desea predecir (columna *Horizonte*) y la frecuencia de muestreo de cada uno (columna *Frecuencia*).

Conjunto	#Series	Frecuencia	H	Misma Long.	Val. Perd.
electricity	321	semanal	8	Si	No
m1	617	mensual	18	No	No
m1	203	cuatrimestral	8	No	No
m3	1428	mensual	18	No	No
m3	756	cuatrimestral	8	No	No
nn5	111	diaria	56	Si	No
nn5	111	semanal	8	Si	No
San Francisco traffic	862	semanal	8	Si	No

Tabla. 3.3: Descripción básica de los conjuntos de datos utilizados. Las columnas #Series, H, Misma Long. y Val. Perd. indican la cantidad de series en el conjunto de datos, el tamaño del horizonte de predicción, si todas las series tienen la misma longitud y si existen valores perdidos ellas respectivamente.

En la tabla 3.4 se muestra que, a pesar de disponer muchas series, ninguna de ellas es larga (tal y como se puede apreciar en la columna *Long. Max*). Recordemos que los algoritmos basados en aprendizaje profundo generan una abstracción a partir de datos, por lo que deberemos entrenar un modelo para cada conjunto en vez de para cada serie (como se hace usando algoritmos estadísticos) para obtener un rendimiento aceptable. Observando la columna `V. Medio`, nos percatamos de que los rangos de valores que maneja cada conjunto de datos es totalmente diferente, y que la diferencia entre los valores máximos y mínimos puede llegar a ser enorme, debido al distanciamiento entre la media de valores mínimos (*V. Med. Min*) y máximos (*V. Med. Max*).

Conjunto	Frecuencia	Long. Min	Long. Max	V. Med. Min	V. Medio	V. Med. Max
electricity	semanal	156	156	1791.942	426778.201	33735152.564
m1	mensual	48	150	0.309	9324.004	1091583.533
m1	cuatrimestral	18	114	1.106	15136.205	640458.780
m3	mensual	66	144	1268.235	4971.283	19331.746
m3	cuatrimestral	24	72	907.697	4983.528	13625.682
nn5	diaria	791	791	7.936	18.404	37.759
nn5	semanal	113	113	55.550	128.825	264.312
San Francisco traffic	semanal	104	104	2.032	9.535	28.418

Tabla. 3.4: Longitud mínima (Long. Min) y longitud máxima (Long. Max) de las series presentes en los conjuntos de datos junto a la media de todos los valores (V. Medio), de los mínimos (V. Med. Min) y de los máximos (V. Med. Max).

Conjunto	Frecuencia	V. Min. Std	V. Std	V. Max. Std
electricity	semanal	547.970	80225.310	12539683.127
m1	mensual	0.041	2760.793	419548.453
m1	cuatrimestral	0.031	2963.876	107121.222
m3	mensual	96.845	1137.075	16445.442
m3	cuatrimestral	39.264	1015.849	5436.546
nn5	diaria	3.323	7.836	15.258
nn5	semanal	9.969	21.642	58.614
San Francisco traffic	semanal	0.075	1.364	9.121

Tabla. 3.5: Desviación típica de todos los valores (V. Std), de los mínimos (V. Std. Min) y de los máximos (V. Std. Max).

Además, todos los conjuntos de datos excepto *San Francisco traffic* poseen valores muy dispares en sus series, tal y como se muestra en la columna *V. Std* de la tabla 3.5. La única desviación relativamente baja se presenta en el dataset mencionado anteriormente, con un valor de 1.364, que acompaña a una media de 9.535. Seguidamente, la desviación de las dos versiones de *nn5* nos informa de que existe dispersión en cierto grado, pero no tanto como en el resto porque son muy altas. Podemos decir que el resto de conjuntos de datos es muy heterogéneo, lo cual es coherente sabiendo que *m3* y *m1* mezclan diferentes dominios entre sus series. Destaca lo sucedido con *electricity*, que a pesar de ser series que registran un mismo suceso, sus estadísticos nos hacen ver que no son homogéneas.

De cara al preprocesamiento es necesario tener en cuenta lo que hemos descubierto gracias a este análisis exploratorio. Tenemos que tener en cuenta la gran diversidad de valores que toman las series, por lo que será idóneo aplicar alguna estrategia que logre suavizar valores anómalos presentes en ellas y, en general, ver cómo hacer frente a estas escalas de valores tan amplias que se dan en la mayoría de conjuntos de datos (sobre todo a la hora de realizar el entrenamiento de los modelos). No será necesario aplicar ninguna estrategia para lidiar con los valores perdidos, ya que no hay.

3.2.2. Preparación de los datos

El preprocesamiento de los datos desempeña un papel fundamental a la hora de conseguir buenos resultados con nuestros modelos. A partir de las conclusiones sacadas en la sección 3.2.1, se ha decidido realizar tres transformaciones a todos los conjuntos de datos.

Identificación y sustitución de valores anómalos

La primera transformación intenta mitigar los valores anómalos que puedan darse en las series debido al gran rango de valores que abarcan. Se ha implementado la heurística expuesta en Martínez et al. [2019] para detectarlos, tomando como definición de valor atípico cuyo valor absoluto es cuatro veces superior a las medianas absolutas de los tres puntos consecutivos anteriores y posteriores a la él. En caso positivo, el valor será sustituido por la media dada a partir de la observación inmediatamente anterior y posterior. En el algoritmo 1 se muestra el pseudocódigo de este paso.

Algoritmo 1: Detección y sustitución de valores anómalos

```
Input: Serie S, Característica c, puntos_consecutivos = 3
Result: Serie transformada S'
S' = {};
for i ∈ {0, ..., longitud(S)} do
    valor = S[i][c];
    punto_anterior = i - puntos_consecutivos;
    punto_siguiente = i + puntos_consecutivos + 1;
    if i ≥ puntos_consecutivos and punto_siguiente < longitud(S) then
        left = series[punto_anterior:i];
        right = series[i+1:punto_siguiente];
        ml = mediana(left, c);
        mr = mediana(right, c);
        check = 4*maximo(abs(ml), abs(mr));
        if abs(val) ≥ check then
            valor = (S[i-1][c] + S[i+1][c])/2;
        end
    end
    S'.añadir(valor);
end
return S';
```

Normalización

Posteriormente, aplicaremos una normalización para reducir el impacto de la escala sobre nuestros modelos profundos. Recordemos que, como se explicó en la sección 2.1.6, la manera de ajustar los pesos de ellos hace uso de la regla de la cadena para calcular cómo varía la función de error en función de la entrada. Si su magnitud es muy

grande, es muy probable que sea la causante de la no convergencia del algoritmo. La ecuación 3.1 expresa cómo realizar una normalización invertible *MinMax* implementada en el paquete `scikit-learn` (Pedregosa et al. [2011]), usada para transformar todos los valores de una serie a la escala $[0, 1]$. Como su nombre indica, usará el valor máximo y mínimo para realizar el escalado, por lo que estos serán sus parámetros *entrenables*.

$$S'' = \frac{S' - min(S')}{max(S') - min(S')} \tag{3.1}$$

Sin embargo, para nuestro caso de uso es necesario redefinir este método. La expresión anterior sería adecuada si para cada serie entrenamos un modelo único. Como hemos visto en el análisis exploratorio, los conjuntos de datos no destacan por las longitudes de sus series. Tal hecho no permitiría la creación de abstracciones que se ajusten adecuadamente a las características de nuestros datos.

Tras varias pruebas, se decidió crear modelos **más generales** que aprendan a partir **todas las series** de un conjunto para así poder adaptarse bien al dominio que las define (ya que, la mayoría de conjuntos de datos provienen de solo uno o de varios que pueden asemejarse) gracias al aumento de ejemplos que logra procesar un modelo en su entrenamiento usando este enfoque. Dicho esto, los máximos y mínimos usados en la ecuación 3.1 no deben ser los de la serie a procesar, sino los **globales** de todo el conjunto de datos (DS). La ecuación 3.2 refleja la normalización a aplicar por cada conjunto de datos. A efectos de implementación, se consigue entrenando un `MinMaxScaler` mediante el método `partial_fit()` de `scikit-learn`.

$$S'' = \frac{S' - min(DS)}{max(DS) - min(DS)} \tag{3.2}$$

Particionamiento y adaptación de los conjuntos de datos

A la hora de abordar un problema de aprendizaje automático necesitamos dividir el conjunto de datos en tres particiones para poder evaluar el rendimiento de nuestro modelo. La primera de ellas será la que nuestro modelo usará durante el proceso de entrenamiento, por lo que se denominará **conjunto de entrenamiento**. A partir de él podemos crear otra partición de **validación** para observar cómo va evolucionando el modelo a medida que es entrenado. Las instancias que no están contenidas en la partición de entrenamiento formarán la de **test**, idóneas para evaluar el modelo porque permiten comprobar la calidad de él al enfrentarse a patrones nunca vistos.

Al abarcar la tarea de predicción de series temporales como un problema de aprendizaje supervisado, es necesario adaptar los datos para construir instancias compatibles con este paradigma. La primera pregunta que nos plantemos en este punto es: ¿Cómo las generamos a partir de una serie?

A diferencia de los modelos estadísticos presentados en el capítulo 2, no podemos alimentar a nuestro modelo con toda la serie de una sola vez. Es necesario dividirla en pequeñas subseries que denominaremos **ventanas**.

Una instancia estará formada por dos ventanas de observaciones ordenadas en el tiempo y sin huecos. La primera representa los valores de entrada que tomará el modelo (X) y la segunda la salida esperada (Y), es decir, las etiquetas. Gracias a esta reestructuración hemos conseguido convertir una serie en un conjunto de pares (x, y) compatible con los modelos supervisados que usaremos en el estudio experimental.

Otro factor a determinar es su tamaño. Para la ventana Y es sencillo, contendrá un número de observaciones igual al tamaño del horizonte de predicción especificado en la tabla 3.3. Sin embargo, ¿cuántos instantes deben de ser suministrados como entrada para lograr predecir dicho horizonte correctamente?

Existen varias alternativas para determinar este número. En Hewamalage et al. [2021] determinaron dos heurísticas adecuadas que consisten en tomar el tamaño del horizonte o el periodo de la estacionalidad incrementado en un 25 % (un cuarto de su valor). Los autores del repositorio recomiendan usar la última opción mencionada, proporcionando una tabla que relaciona la frecuencia de muestreo con el periodo buscado (tabla 3.6).

Frecuencia	**Periodo**
Diaria	7 observaciones
Semanal	52 observaciones
Mensual	12 observaciones
Cuatrimestral	4 observaciones

Tabla. 3.6: Periodo de la serie en función de la frecuencia de muestreo. Para las series con frecuencia diaria, se establece un periodo de una semana (7 días). Para el resto, se establece el periodo de un año (52 semanas = 12 meses = 4 cuatrimestres = 1 año).

En la tabla 3.7 presentamos los tamaños de ventana que se usarán para cada conjunto de datos junto a su frecuencia, periodo y horizonte. Apreciamos la aparición de un fenómeno que puede parecer incoherente, ¿por qué para los conjuntos *m1*, *m3* y *nn5* con frecuencia diaria el tamaño de la entrada es menor que el horizonte de predicción? Como se puede intuir, es consecuencia de la estrategia usada. Queremos que el modelo observe como mínimo un periodo estacional completo con un leve ruido que

le permita también modelar estos comportamientos teniendo en cuenta la longitud mínima que presentan los conjuntos de datos. Si realizamos un aumento en el tamaño, reduciremos el número de ventanas generadas, impidiendo incluir efectivamente patrones contenidos en series cortas presentes en los conjuntos de entrenamiento. De todos modos, la determinación de este tamaño no es universal (como se ha comprobado), por lo que en función de nuestro caso de uso será necesario ajustarlo según las características que presenten nuestros datos.

Conjunto	Frecuencia	Periodo	Tam. Entrada	Horizonte
electricity	semanal	52	65	8
m1	mensual	12	15	18
m1	cuatrimestral	4	5	8
m3	mensual	12	15	18
m3	cuatrimestral	4	5	8
nn5	diaria	7	8	56
nn5	semanal	52	65	6
San Francisco traffic	semanal	52	65	8

Tabla. 3.7: Conjuntos de datos con su periodo, tamaño de ventana y horizonte de predicción.

Nuestro objetivo es lograr lo ilustrado en la figura 3.3, recorrer la serie para ir formando los pares mencionados anteriormente tal y como describe el algoritmo 2.

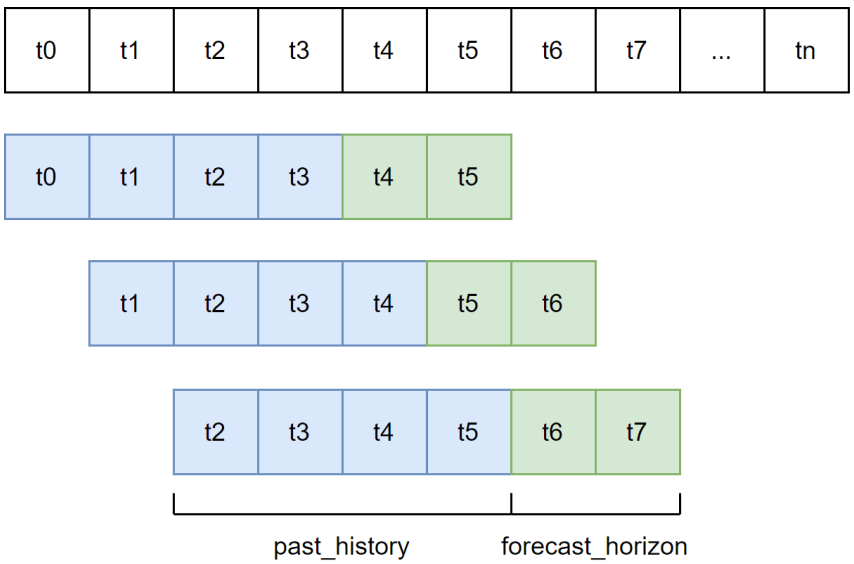

Figura 3.3: Generación de ventanas deslizantes con tamaño de entrada (`past_history`) 4 y horizonte de predicción (`forecast_horizon`) $h = 2$. El tamaño total de una ventana completa equivale a la suma del horizonte de predicción con el número de instantes usados como entrada. Fuente: Elaboración propia.

Algoritmo 2: Construcción de ventanas deslizantes

Input: ds: Conjunto de datos, h: Horizonte de predicción, $past\_history$: Tamaño de la ventana
Result: Tupla (X, Y)
X = {};
Y = {};
for *serie* $\in DS$ **do**
  for *i* $\in \{past\_history, ..., longitud(serie) - h + 1\}$ **do**
    x = serie[i - $past\_history$:i];
    y = serie[i:i + h];
    X.añadir(x);
    Y.añadir(y);
  end
end
return (X, Y);

Recorremos cada serie del conjunto de datos empezando por el instante que coincide con el tamaño de la ventana de entrada, permitiendo que la primera contenga los primeros `past_history` instantes, que se usarán para predecir los `forecast_horizon` siguientes. A medida que recorremos la serie, iremos desplazando en una observación la ventana, hasta llegar al instante cuyo horizonte de predicción esperado contenga el último valor de la serie. Al final, tendremos un total de ventanas equivalente a $longitud(serie) - past\_horizon - forecast\_horizon$. Todas estas ventanas se guardan en las listas X e Y, de tal manera que la ventana i almacene sus instantes de entrada en y su horizonte esperado usando el mismo índice.

Para cada serie, la partición de **entrenamiento** contendrá todas las ventanas generadas, excepto la última que formará el conjunto de **test**. En la figura 3.4 se muestra un ejemplo de los instantes de una serie usados para entrenar el modelo, los valores de entrada de la instancia de prueba y su salida esperada. Consecuentemente, la unión de todas las particiones de entrenamiento de todas las series formará el conjunto de entrenamiento de un conjunto de datos e ídem para el de prueba.

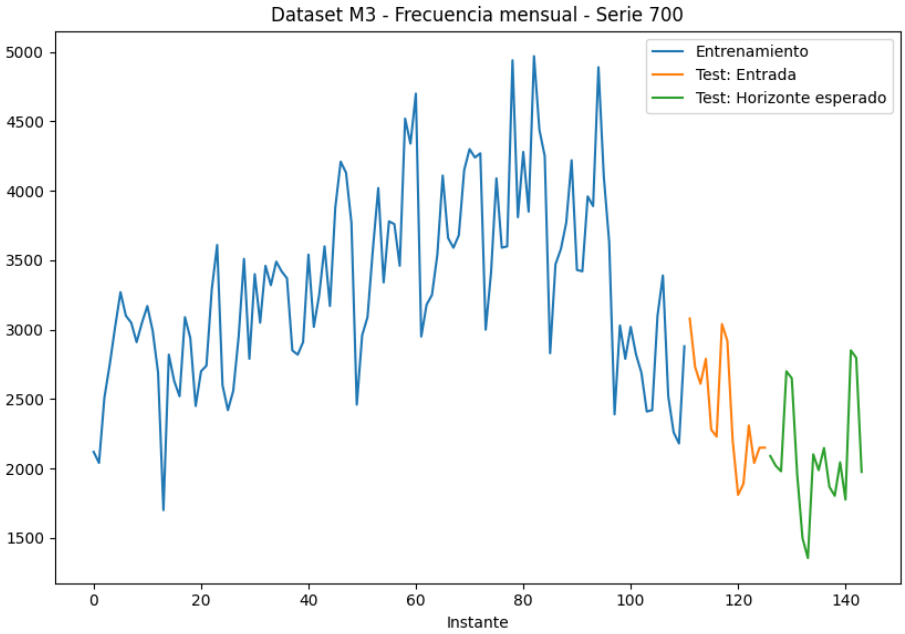

Figura 3.4: División de la serie número 700 del conjunto M3 con frecuencia mensual. Se detallan las observaciones que componen la entrada y la salida esperada del conjunto de prueba. Fuente: Elaboración propia.

3.3. Implementación de los modelos

En la siguiente sección describiremos cómo se han implementado los modelos recurrentes y el modelo convolucional descritos en 2.1.7) usando las herramientas seleccionadas. Antes de comenzar mostraremos el proceso general que se debe de seguir para construir un modelo en *Pytorch* junto a *Pytorch Lightning*.

Todos los modelos neuronales implementados deberán de heredar de `Module`, clase base localizada dentro del paquete `nn`. Ella nos permite dotar de consistencia a todas las implementaciones realizadas, obligando al desarrollador a sobrescribir la función `forward()` para implementar el paso hacia adelante y asegurando que todos nuestros modelos poseen los atributos y funciones necesarias para que *PyTorch* los pueda manejar correctamente. En el listado 3.1 se muestra un modelo de ejemplo.

```
1 class MiModeloModule(nn.Module):
2     def __init__(self):
3         super().__init__()
4         # Inicialización de atributos aquí
5         pass
6     def forward(self, x):
7         # Paso hacia adelante, procesar datos de entrada y emitir la salida del modelo
8         pass
```

Listado 3.1: Implementación de un modelo a partir de `nn.Module`

En este momento podríamos entrenar ya nuestro modelo para posteriormente usarlo. Previamente, tendremos que implementar el bucle de entrenamiento, seleccionando el optimizador a usar y transfiriendo aquello que sea necesario de CPU a GPU y viceversa. *PyTorch Lightning* nos ahorrará la implementación de todos estos códigos repetitivos mediante la integración de entrenadores (entre otras cosas) encargados de realizar todo lo anterior. Nosotros solo tenemos que encargarnos de crear una nueva clase que herede de `pl.LightningModule` que contenga como atributo al modelo anterior y redefinir aquellos disparadores que consideremos necesarios. En el listado 3.2 se muestra la estructura base de un modelo usando *PyTorch Lightning*.

En el constructor de estos módulos es esencial especificar como atributos los **modelos de *Pytorch*** implementados, sus hiperparámetros específicos (como el número de capas), el ratio de aprendizaje del optimizador, la función de pérdida, horizonte de predicción, tamaño de la ventana de entrada o cualquier otro elemento que sea necesario para su funcionamiento o monitorización (como listas o diccionarios). Especificaremos también las métricas a usar mediante el uso de la librería de *torchmetrics*, incluida *PyTorch Lightning*.

Para cada etapa por la que pasa un modelo en su evaluación existen funciones cuya cabecera sigue el patrón `on_ETAPA_start()` y `on_ETAPA_end()` que se ejecutan justo antes o después del inicio o final de ella. También podemos encontrar otras como `on_ETAPA_epoch_start()` y `on_ETAPA_epoch_end()`, de similar funcionamiento para cada época que comprende la etapa. El uso de todos estos métodos permitirán realizar un seguimiento del modelo completo y personalizado a los intereses del desarrollador.

Los procedimientos `ETAPA_step()` contendrán la llamada al paso hacia adelante del modelo implementado y el cálculo de la pérdida, que será usada posteriormente por el optimizador seleccionado mediante `configure_optimizers()` para realizar el paso hacia atrás de forma automática.

```
class MiModelo(pl.LightningModule):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Hiperparámetros del modelo, y todo lo especificado anteriormente.
        self.model = MiModelo() # Modelo anterior

    # ====== ENTRENAMIENTO =======
    def on_train_start(self):
        # Al inicio del entrenamiento, se ejecuta.
        # Útil para inicializar estructuras que contendrán estadíticas del entrenamiento.
```

```
11
12      def on_train_epoch_end(self):
13          # Al finalizar cada época del entrenamiento, se ejecuta.
14          # Podemos usarlo para visualizar estadísticas en Tensorboard.
15
16      def training_step(self, batch, batch_idx):
17          # Paso de entrenamiento, se han dejado los pasos generales.
18          # Si es necesario se puede procesar la entrada o la salida del modelo.
19          x, y = batch
20          pred = self.model(x) # Equivale a hacer model.forward(x)
21          loss = calcular_loss(pred, x)
22          return loss # Simpre se debe devolver
23
24      # ====== VALIDACIÓN =======
25      def on_validation_epoch_end(self):
26          # Al finalizar de procesar una época del conjunto de validación, ejecutar está función.
27          # Misma finalidad, visualizar estadísticas
28          pass
29
30      def validation_step(self, batch, batch_idx):
31          # Paso de validación. Similar al de entrenamiento.
32          pass
33
34      # ====== TEST =======
35      # Paso de predicción (test).
36      def test_step(self, batch, batch_idx):
37          # Realizar inferencia del conjunto de test, calcular métricas de evaluación
38          pass
39
40      # ===== PRODUCCION =====
41      # Devolver predicciones
42      def predict_step(self, batch, batch_idx: int, dataloader_idx: int = 0):
43          # Realizar inferencia a partir del modelo
44          pass
45
46      # ===== CONFIGURACIÓN DEL MODELO =====
47      # Optimizador a usar, siempre usaremos ADAM
48      def configure_optimizers(self):
49          optimizer = optim.Adam(self.parameters(), lr=self.lr)
50          return optimizer
```

Listado 3.2: Adaptación de un modelo de *PyTorch* a *PyTorch Lightning*.

Los hiperparámetros que debe de recoger esta clase son (entre los propios del modelo que instancia) el ratio de aprendizaje del optimizador (`lr`) y la función de pérdida a usar (`loss`). La función de pérdida a usar será `nn.SmoothL1Loss` en todos los modelos, similar a la función MAE pero siendo menos sensible a los valores anómalos y permitiendo evitar la explosión o desvanecimiento de gradiente en algunos casos (PyTorch [2023]).

Resumidamente, los módulos que heredan de `nn.Module` contendrán la lógica de funcionamiento y sus hiperparámetros deberán de estar parametrizados para que su módulo `pl.LightningModule` pueda inicializarlos correctamente, gestionando así su

entrenamiento, validación y testeo mediante el uso de un objeto entrenador `pl.Trainer`. En él se especificará el máximo de épocas a usar (parámetro `max_epochs`), si se desea usar GPU para acelerar los cálculos (`accelerator`) e incluso permite usar estrategias de entrenamiento distribuido y multi-GPU (`strategy`). El listado 3.3 muestra cómo realizar entrenamientos de manera muy sencilla.

```
# Creamos nuestro modelo de Lightning
modelo = MiModelo()
# Creamos el entrenador de Lightning
trainer = pl.Trainer(max_epochs=50, accelerator="gpu")
# Entrenamos el modelo
trainer.fit(modelo, train_dataloaders=train_dl)
# Probamos nuestro modelo
metricas = trainer.test(modelo, dataloaders=test_dl)
```

Listado 3.3: Entrenamiento de un modelo usando *PyTorch Lightning*.

Una vez que sabemos las base de todos los modelos, procederemos a detallar algunos aspectos de implementación propios de ellos en las siguientes secciones.

3.3.1. Redes recurrentes

El paquete `nn` de *PyTorch* suministra una implementación de las celdas recurrentes LSTM y GRU (`nn.LSTM` y `nn.GRU`), con la capacidad de apilar varias nativamente. Si consultamos su documentación apreciamos que los hiperparámetros que presentan son muy similares, permitiendo parametrizar en nuestro módulo el tipo de celda a usar sin necesidad de preocuparnos por ellos.

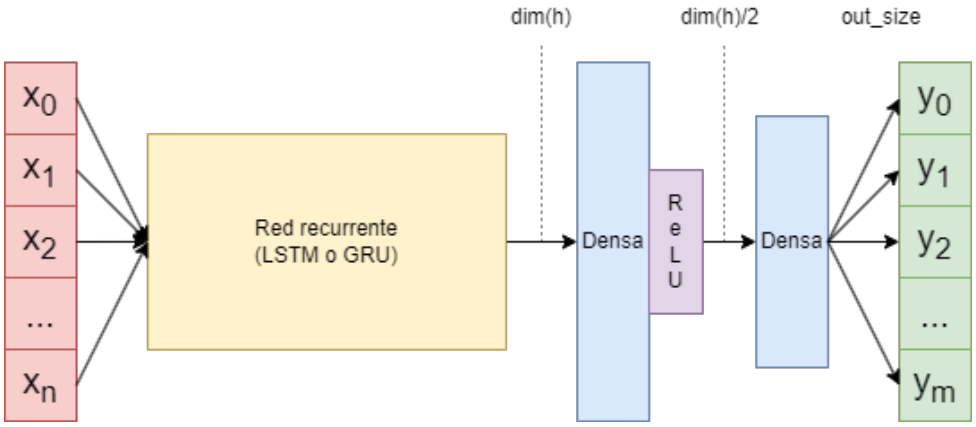

Figura 3.5: Arquitectura general de las redes recurrentes con dos capas de proyección, detallando las características de entrada y salida de cada una. Fuente: Elaboración propia.

Tal y como se explica anteriormente, crearemos dos clases para implementar la lógica y gestión del modelo. `MIMORNNModel` contendrá las capas recurrentes necesarias

junto a dos capas de proyección, necesarias para realizar la predicción multisalida (figura 3.5). La primera de ellas poseerá una función de activación ReLU mientras que la segunda usará la función identidad ($f(x) = x$), agregando el estado oculto de entrada, reduciendo su dimensión a la mitad para posteriormente volver a agregarlo y emitir el horizonte de predicción deseado. Más adelante detallaremos este paso.

El módulo de *PyTorch Lightning* `MIMORNN` instanciará la clase anterior para controlar su uso, por lo que será necesario declarar los siguientes atributos en su constructor para inicializar correctamente los hiperparámetros del modelo:

- `num_rnn_layers` (`int`): **Número de capas** de la red neuronal recurrente, es decir, número de celdas recurrentes apiladas.
- `hidden_dim` (`int`): **Tamaño del estado oculto** h de una celda.
- `input_features` (`int`): **Número de características** presentes en cada paso temporal de la secuencia de entrada. No debemos de confundirla con la **longitud** de la secuencia.
- `input_size` (`int`): **Longitud** de la secuencia de **entrada**.
- `output_size` (`int`): **Longitud** de la secuencia de **salida**, es decir, tamaño del horizonte que se desea predecir. Usado en la capa de proyección.
- `dropout` (`float`): Indicar qué porcentaje de celdas queremos desactivar en el entrenamiento de manera aleatoria. En nuestro proyecto siempre será 0.
- `model_name` (`str`): Nombre del modelo.
- `return_sequences` (`bool`): Intenta imitar el comportamiento del hiperparámetro booleano `return_sequences` implementado en las capas recurrentes de *Tensorflow*. Determina si el modelo debe utilizar todos los estados ocultos generados en la última capa de la red recurrente, en lugar de solo el correspondiente al último paso, al realizar la proyección.
- `flavor` (`str`): **Tipo de celda** a usar (LSTM o GRU).

Debido al uso de dos estados ocultos, el módulo `nn.LSTM` devuelve tres tensores en vez de dos. El primero de ellos (`output`) contiene todos los estados ocultos para cada instante de cada secuencia del lote devueltos por la última capa de la red. La tupla `(hn, cn)` recoge el último estado oculto y de la celda que sale de cada capa de la red para cada secuencia. Para `nn.GRU` se emplea la misma lógica, pero eliminando el estado `cn`. La figura 3.6 muestra mejor el valor que toman estos tensores.

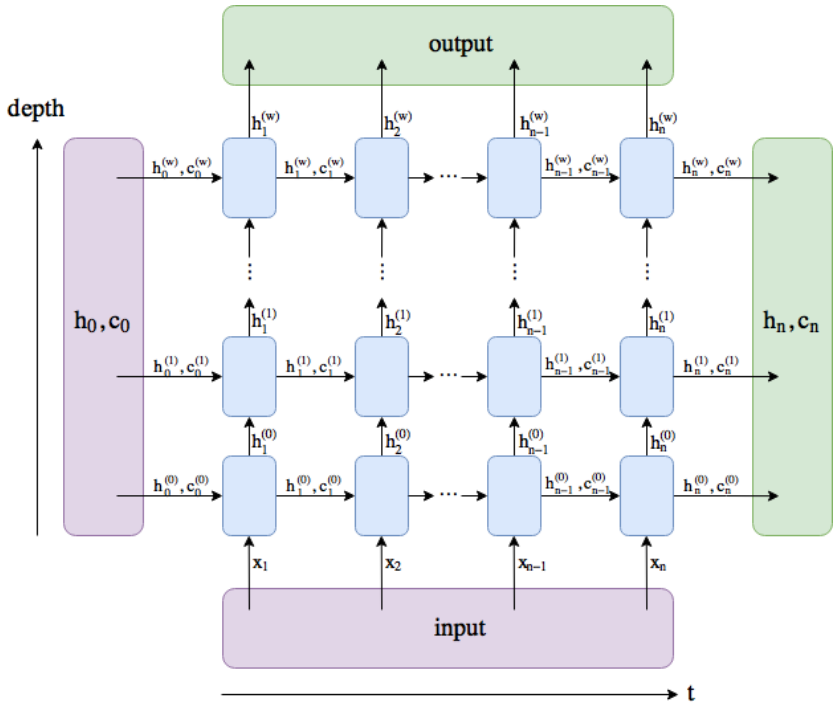

Figura 3.6: Propagación de estados de una red recurrente celdas LSTM apiladas. Una fila representa una celda desenrollada. En el caso de celdas GRU, debemos obviar los estados c_n. Fuente: StackOverflow [2018].

A partir de la ilustración anterior, observamos que si `return_sequences` es verdadero, la salida del modelo dependerá de `input_size` estados ocultos y no solo del último. Por lo tanto, antes de aplicar las capas de proyección es necesario concatenar todos estos estados en un vector fila (lo que se denomina **aplanar**), obteniendo así una salida de `input_size` · `hidden_dim` características que se usarán para determinar el horizonte de predicción (`output_size`) deseado.

Por último, para las celdas LSTM se han inicializado todos los pesos siguiendo una distribución normal. Los sesgos de la puerta de olvido se fijan a 1 para evitar olvidar grandes cantidades de información en épocas tempranas del entrenamiento. Para ello, ha sido necesario iterar por las estructuras que contienen los parámetros del modelo y determinar qué índices se corresponden con el que se desea cambiar. Para la distribución normal se ha usado la utilidad `nn.init.kaiming_normal_` del *framework* (listado 3.4).

```
def init_weights(self):
    for names in self.rnn._all_weights:
        if isinstance(self.rnn, nn.LSTM):
            for name in filter(lambda n: "bias" in n, names):
                bias = getattr(self.rnn, name)
                n = bias.size(0) # Tamaño de la matriz
                start, end = n//4, n//2 # Rango donde se situan los sesgos
                bias.data[start:end].fill_(1) # Forget a 1, para no olvidar tanto al principio
```

```
9
10              for name in filter(lambda n: "weight" in n, names):
11                  weight = getattr(self.rnn, name)
12                  nn.init.kaiming_normal_(weight.data, mode='fan_in', nonlinearity='relu')
```

Listado 3.4: Inicialización de pesos de celdas LSTM.

Para ilustrar mejor la organización de las clases correspondientes se adjunta el diagrama de clases de las descritas en este apartado (figura 3.7).

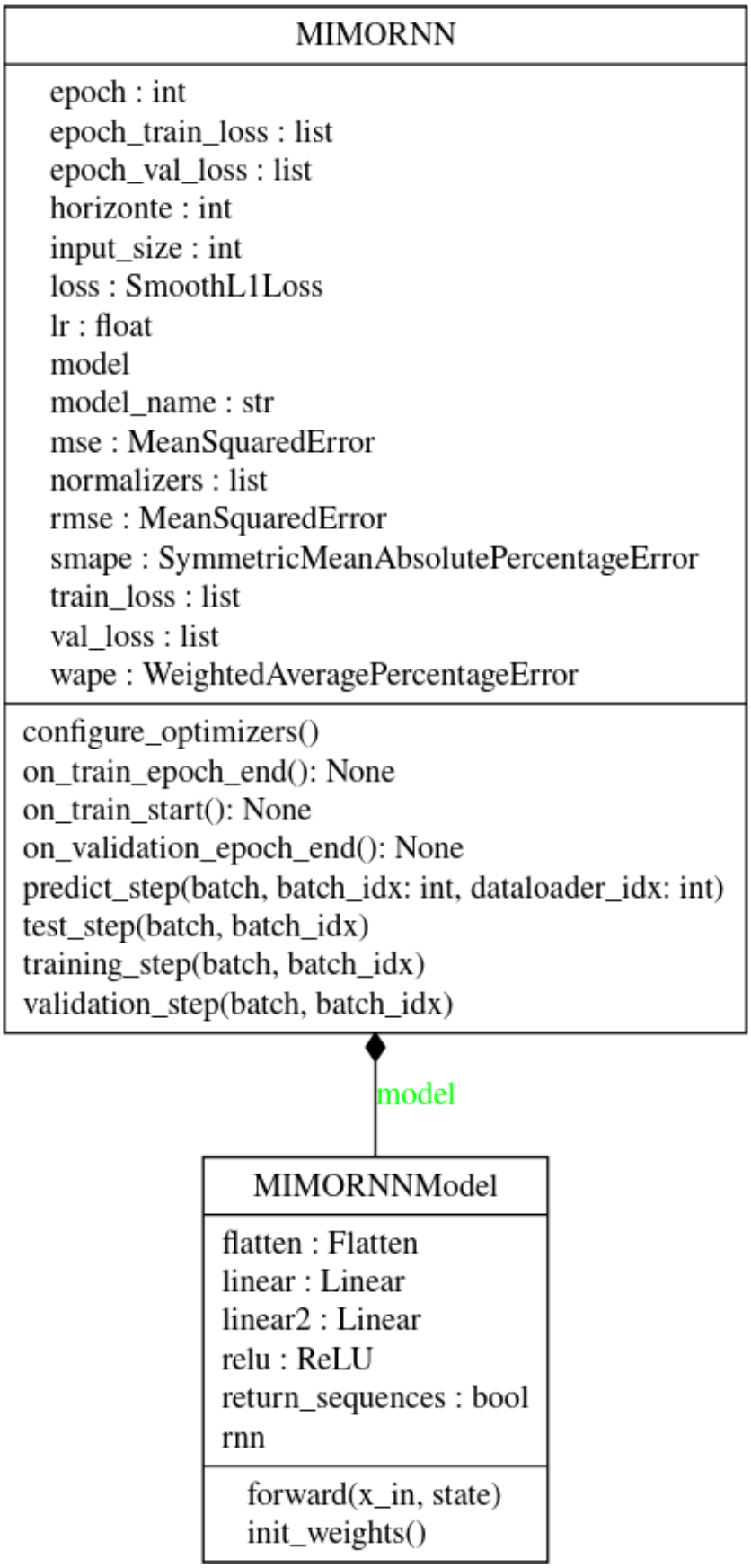

Figura 3.7: Diagrama de clases de la implementación de las redes recurrentes. Fuente: Elaboración propia.

3.3.2. Redes convolucionales temporales

La implementación de una TCN es más compleja que las de las redes recurrentes anteriores. Tal y como se describe en la sección 2.3.3, es necesario definir correctamente las **convoluciones causales dilatadas** para poder formar **bloques recurrentes** con ellas y componer así nuestra red.

El módulo `nn.Conv1D` permite la implementación de diferentes capas convoluciones unidimensionales en función de sus hiperparámetros. Será de nuestro interés saber que los atributos `padding`, `dilatation` y `kernel_size` permiten especificar el desplazamiento de la entrada, factor de dilatación y tamaño del filtro de las convoluciones que la conforman. Hay que tener en cuenta que este módulo añade valores nulos en ambos lados de la entrada cuando a nosotros solo nos interesa por el principio, por lo que es necesario eliminar las posiciones finales sobrantes tras aplicar la convolución.

Se ha implementado una utilidad denominada `Chomp1D` para ello, que elimina las `chomp_size` posiciones finales de un tensor (figura 3.8). La construcción de filtros y la aplicación de la separación de las entradas dada por el factor de dilatación se realiza automáticamente, por lo que no es necesario modificar nada más.

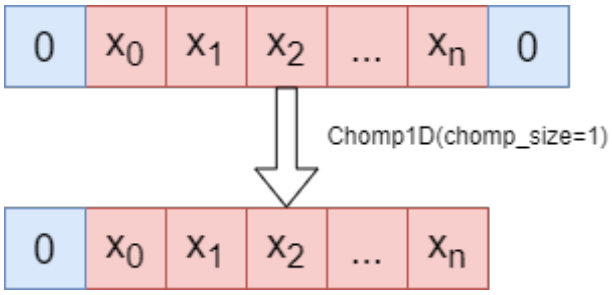

Figura 3.8: Aplicación de `Chomp1D` a un tensor desplazado por *PyTorch* en 1 unidad. Fuente: Elaboración propia.

Ya sabemos que *PyTorch* nos permite la implementación de las convoluciones que buscamos. Para implementar los bloques residuales que compondrán nuestra red crearemos el módulo `TemporalBlock`, siendo necesario especificar los siguientes atributos en su constructor:

- `n_inputs` (`int`): Número de canales de **entrada** (se corresponde con el número de características).
- `n_outputs` (`int`): Número de canales de **salida** que producirá la convolución.
- `kernel_size` (`int`): Tamaño del **kernel** de la convolución.

- `stride` (`int`): Tamaño del **paso** en la convolución.
- `dilation` (`int`): Factor de **dilatación** para la convolución.
- `padding` (`int`): Tamaño del relleno en la convolución, usado para **desplazar** la entrada.
- `dropout` (`float`): Igual que en la sección anterior, porcentaje de celdas queremos desactivar en el entrenamiento de manera aleatoria.

La arquitectura de estos bloques seguirá el esquema de la figura 2.24 haciendo uso de `Chomp1D` y`nn.Conv1D` para las convoluciones del bloque, `nn.Dropout` para evitar el sobreajuste y `nn.utils.weight_norm` para aplicar la normalización de los pesos presente después de cada convolución. La conexión residual será simplemente una suma entre la entrada y la salida de la última convolución, aplicando previamente una convolución 1D con tamaño de kernel de 1 unidad (sin *padding* ni dilatación) a la entrada para igualar el número de canales con la salida del bloque si es necesario.

Ahora, podemos construir nuestra TCN como una secuencia de bloques recurrentes. La clase `TemporalConvNetModel` se encargará de ello, siendo necesario especificar los siguientes atributos en su constructor para propagarlos a los bloques recurrentes:

- `input_feat` (`int`): Número de características que poseen los pasos de la secuencia de entrada, es decir, el número de canales de entradas del primer bloque.
- `tcn_hidden_size` (`int`): Tamaño oculto de TCN, se corresponde con el número de canales de salida del bloque recurrente.
- `tcn_n_layers` (`int`): Número de bloques recurrentes a apilar.
- `tcn_dropout` (`float`): Tasa de *dropout* en TCN.
- `out_size` (`int`): Tamaño del horizonte de predicción a emitir por el modelo a partir de la información recogida por la TCN. Parámetro usado en una capa lineal sin función de activación para realizar la proyección deseada.

El factor de dilatación y el tamaño de la convolución que se usarán no se especifican en el constructor anterior, aunque la implementación realizada permite fácilmente su parametrización. Se ha considerado fijar el primero como una potencia de base 2 y exponente igual al índice del bloque (2^i), mientras que el segundo siempre será igual a 2. Aclarar que el desplazamiento a realizar sobre la entrada de cada convolución no

será $tam\_nucleo - 1$ unidades porque hay que tener en cuenta el efecto de la dilatación, así que deberá ser $(tam\_nucleo - 1) \cdot 2^i$. En el algoritmo 3 se muestra el proceso de construcción de una TCN.

Algoritmo 3: Construcción de una TCN

Input: input_feat, tcn_hidden_size, tcn_n_layers, tcn_dropout, out_size.
Result: Red TCN con capa de proyección
num_channels = crear_lista_inicializada(tcn_hidden_size, tcn_n_layers);
kernel_size = 2;
layers = {};
num_levels = len(num_channels);
for *i* ← *0* **to** *num_levels - 1* **do**
    dilation_size = 2^i;
    in_channels = input_feat si i = 0 else num_channels[i-1];
    out_channels = num_channels[i];
    layers.añadir(TemporalBlock(in_channels, out_channels, kernel_size,
     stride=1, dilation=dilation_size, padding=(kernel_size-1) · dilation_size,
     dropout=tcn_dropout));
end
network = crear_modelo_secuencial(layers);
out_proj = lineal(tcn_hidden_size, out_size);
return *(network, out_proj)*

Por último, la clase `TemporalConvNet`, al igual que `MIMORNN`, será el módulo de *PyTorch Lightning* qué instancia la clase anterior para hacer uso de ella de cada al entrenamiento y prueba de nuestro modelo. Los parámetros de su constructor son exactamente iguales que los de `TemporalConvNetModel`, añadiendo el nombre del modelo. En la figura 3.9 se muestra el diagrama de clases correspondiente con la implementación de este modelo.

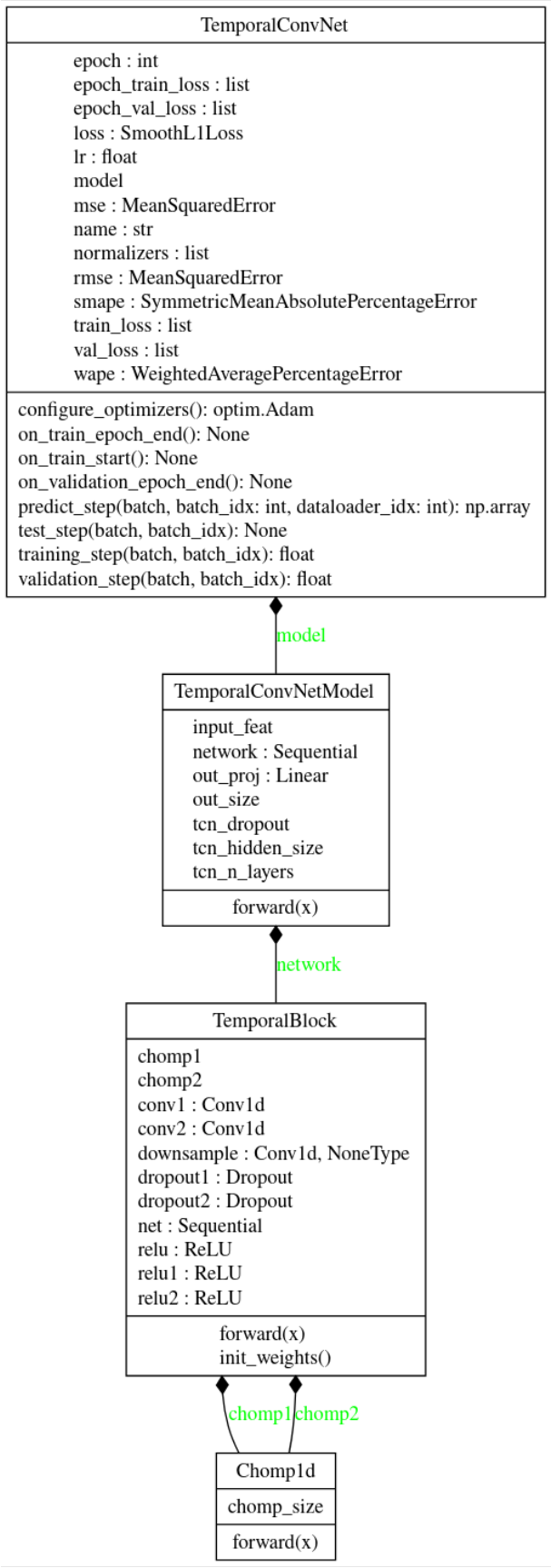

Figura 3.9: Diagrama de clases de la implementación de las TCN. Fuente: Elaboración propia.

3.4. Marco experimental

En esta sección se presentan diferentes aspectos que se deben de tener en cuenta antes de realizar la experimentación, como la finalidad del estudio, las métricas usadas y los recursos hardware disponibles para su realización.

3.4.1. Finalidad del estudio

El objetivo del estudio es evaluar los modelos implementados anteriormente usando diferentes arquitecturas para comprender su funcionamiento ante los 8 conjuntos de datos descritos en la sección 3.2.1. Dispondremos de varias medidas que nos ayuden a compararlos en términos de rendimiento, tiempo de entrenamiento y tiempo de inferencia.

model	épocas	capas	dim_h	bs	lr	%dropout
RNN-LSTM	50, 100, 200	1, 2, 4	32, 64, 128	16, 32	1e-3, 1e-2	0
RNN-GRU	50, 100, 200	1, 2, 4	32, 64, 128	16, 32	1e-3, 1e-2	0
TCN	50, 100, 200	1, 2, 4	16, 32, 64	16, 32	1e-3, 1e-2	0, 0.1, 0.25

Tabla. 3.8: Hiperparámetros a usar en la experimentación para cada modelo. *dim_h* hace referencia al tamaño del estado oculto en las redes recurrentes y al número de canales generados por las convoluciones en la TCN. *bs* indica el tamaño del lote y *lr* la tasa de aprendizaje del optimizador.

Para cada conjunto de datos se realizará un total de **540 pruebas** (216 correspondientes a los modelos recurrentes y 324 a la TCN). Cada configuración a probar viene dada al combinar los diferentes hiperparámetros presentes en la tabla 3.8. Como contamos con 8 conjuntos, se entrenará y probará un total de **4320 modelos** diferentes. Esta exhausta experimentación podrá esbozar cómo se comporta cada modelo en función de su configuración, resolviendo preguntas como: ¿Cómo afecta la naturaleza de los datos al modelo?, ¿en qué punto conviene no aumentar la complejidad del modelo? O ¿existen diferencias significativas entre diferentes algoritmos?

Para asegurar la reproducibilidad de la experimentación, los *scripts* relativos a la experimentación contienen mecanismos para inicializar cualquier mecanismo estocástico en función de una semilla determinada. Se han seguido indicaciones contenidas en la documentación de *PyTorch* para ello.

3.4.2. Evaluación de modelos

La evaluación de los modelos será fuera de línea (*offline*) usando métricas que podemos separar según la característica que evalúen del modelo. Tres de ellas nos permitirán saber la **calidad** de las predicciones emitidas, mientras que las otras dos informarán sobre el **tiempo** que se tarda en entrenar y predecir. Para cada conjunto de datos, los resultados se han guardado en ficheros CSV organizados según la siguiente cabecera (listado 3.5).

```
1 DATASET;MODEL;LAYERS;UNITS;EPOCHS;BS;LR;RMSE;SMAPE;WAPE;TRAIN_TIME_NS;INFERENCE_TIME_NS
```

Listado 3.5: Cabecera de un fichero CSV que recoge los resultados de la experimentación.

Métricas para evaluar la calidad de las soluciones

Como métricas para evaluar la calidad se han seleccionado la raíz del error cuadrático medio (RMSE, ecuación 3.3), el error porcentual absoluto medio simétrico (SMAPE, ecuación 3.4) y el error porcentual medio ponderado (WAPE, ecuación 3.5).

Hay que tener en cuenta que el RMSE es **absoluto**, mientras que el resto de métricas son **relativas**. Por lo tanto, **no** podremos usar el RMSE para comparar el rendimiento de nuestros modelos en conjuntos de datos diferentes.

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{t=0}^{n}(y_t - \hat{y}_t)^2} \tag{3.3}$$

$$SMAPE = \frac{2}{n}\sum_{t=0}^{n}\frac{|y_t - \hat{y}_t|}{\max(|y_t| + |\hat{y}_t|, 117 \cdot 10^{-8})} \tag{3.4}$$

$$WAPE = \frac{\sum_{t=0}^{n}|y_t - \hat{y}_t|}{\sum_{t=0}^{n} y_t} \tag{3.5}$$

A la hora de analizar los resultados interpretaremos el RMSE como la desviación media de las diferencias entre los valores de datos observados y los valores predichos del modelo. El SMAPE y el WAPE son modificaciones de otra métrica (MAPE) y tienen en cuenta la escala de los datos. La primera de estas modificaciones dota de [3]simetría y acota el error relativo cometido en el intervalo, $[0, 2]$ mientras que la segunda reescala

[3]En este contexto: $SMAPE(a, b) = SMAPE(b, a)$

el error en función de los valores reales para hacerlo comparable entre series que poseen escalas variadas (como puede ocurrir en varios conjuntos de datos descritos anteriormente).

Métricas para determinar el tiempo de entrenamiento e inferencia

De cara a los tiempos, simplemente se han medido en **nanosegundos** a la hora de ejecutar cada una de las pruebas definidas. Es por ello, que dependen de la máquina en la que se hayan ejecutado y pueden variar si se ejecuta en otra. Como ocurre con el RMSE, los tiempos son métricas **absolutas**. A la hora de realizar el análisis serán convertidos a segundos para facilitar su comparación e interpretación.

3.4.3. Recursos *hardware*

Para la realización de los experimentos, se cuenta con un clúster del Instituto Andaluz de Inteligencia Artificial y Ciencias de Datos (DasCI), localizado en el edificio C6 de la Universidad de Jaén. Los recursos del mismo son más que suficientes. Se presentan a continuación:

- **CPU**: 2 x Intel(R) Xeon(R) CPU E5-2698 v4, Frecuencia de reloj de 2.20GHz.
- **GPU**: 8 x Tesla V100 SXM2 con 32GB VRAM.
- **Memoria RAM**: 512 GB.
- **Almacenamiento**: Discos en RAID 5 con un total de 5.2TB.

Capítulo 4

RESULTADOS

Tras aplicar los métodos implementados a los materiales, se procede a analizar los resultados obtenidos. Empezaremos con un análisis de los tiempos de entrenamiento e inferencia y posteriormente comentaremos la calidad de las predicciones obtenidas según su RMSE, SMAPE y WAPE. Nos apoyaremos en el uso de diagramas de bigotes para visualizar eficazmente la experimentación realizada. Aun así, si se desea obtener un mejor detalle de ella, se podrá consultar en el Anexo A.

4.1. Análisis de los tiempos de entrenamiento e inferencia

El tiempo de entrenamiento se define como el tiempo que tarda un modelo en construir una abstracción del conjunto de datos y variará en función de la cantidad de datos disponibles y la arquitectura del modelo. Por otra parte, el tiempo de inferencia es el tiempo que tarda un modelo en realizar una predicción desde que empieza a procesar la entrada, por lo que no llegará a ser tan alto como el de entrenamiento. En las figuras 4.1 y 4.2 se muestran los diagramas de bigotes para cada modelo en todos los conjuntos de datos usados.

Tras analizar los resultados obtenidos observamos que existe una gran desviación de los tiempos de entrenamiento para todos los conjuntos de datos (tablas presentes en la sección A.1), sobre todo en los modelos TCN al poseer más valores anómalos. Destacar que, aunque en términos de media y desviación los modelos recurrentes superen a la TCN, los tiempos de entrenamiento mínimos no, lo que nos informa de que el modelo convolucional es más sensible a cambios sobre su arquitectura que el resto de modelos. Respecto al uso de *dropout* en estos modelos, no se aprecian diferencias significativas.

Los tiempos de inferencia son adecuados, ninguna ejecución supera el segundo, siendo las redes TCN las portadoras de los tiempos más bajos. El principal motivo es la paralelización intrínseca que posee al poder procesar la entrada en un paso sin necesidad de iterar por todos los instantes individualmente. En la sección A.2 se puede apreciar con mayor detalle lo comentado.

Respecto al uso de celdas LSTM o GRU, tampoco se aprecian diferencias significativas, aunque el tiempo medio de entrenamiento sea un 11 % mayor en el conjunto *m1* con frecuencia mensual para las celdas GRU respecto a las LSTM, poseedoras también de una levemente mayor desviación en la mayoría de conjuntos.

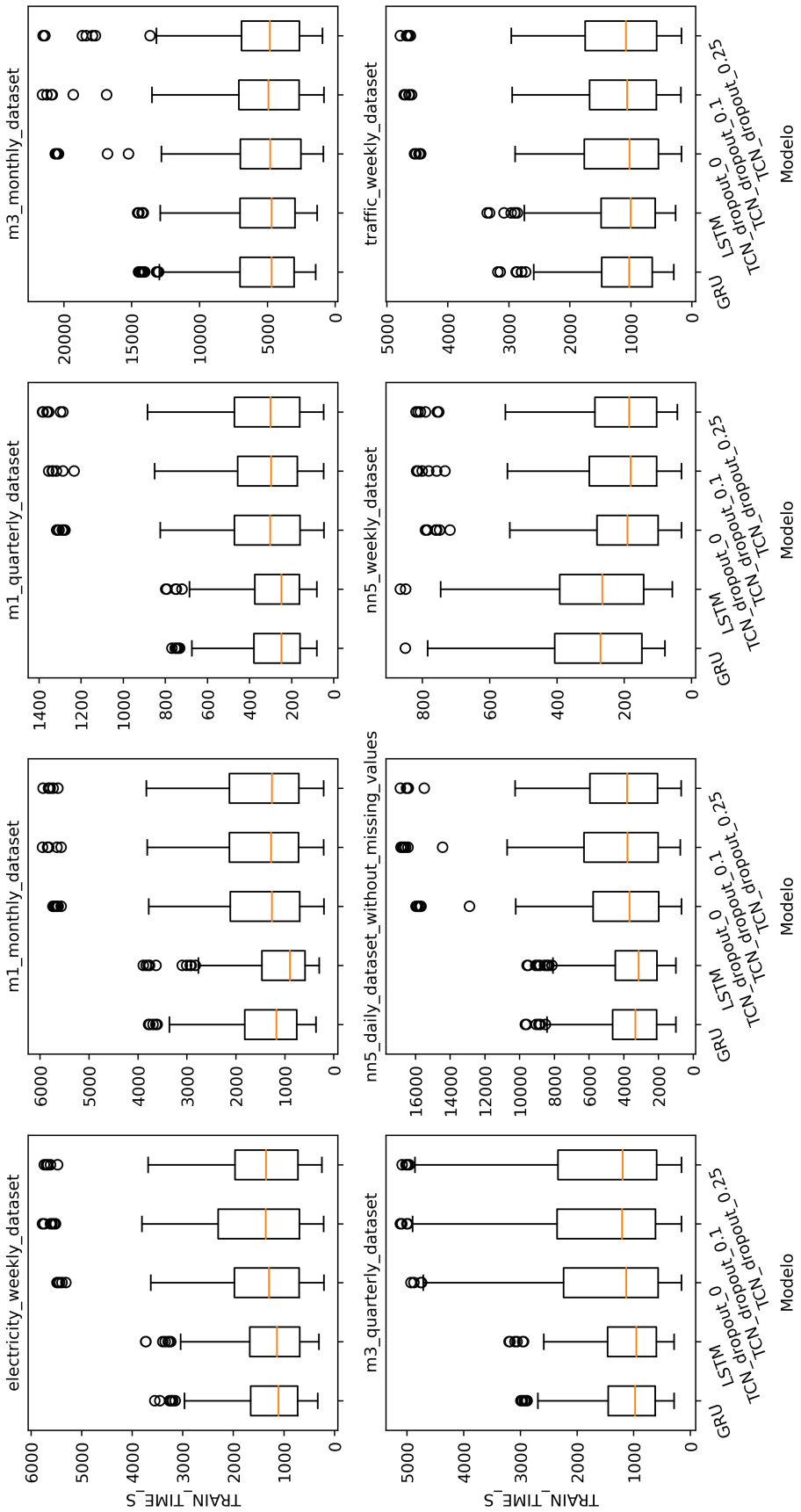

Figura 4.1: Diagramas de cajas y bigotes del tiempo de entrenamiento (en segundos) para cada modelo y conjunto de datos.

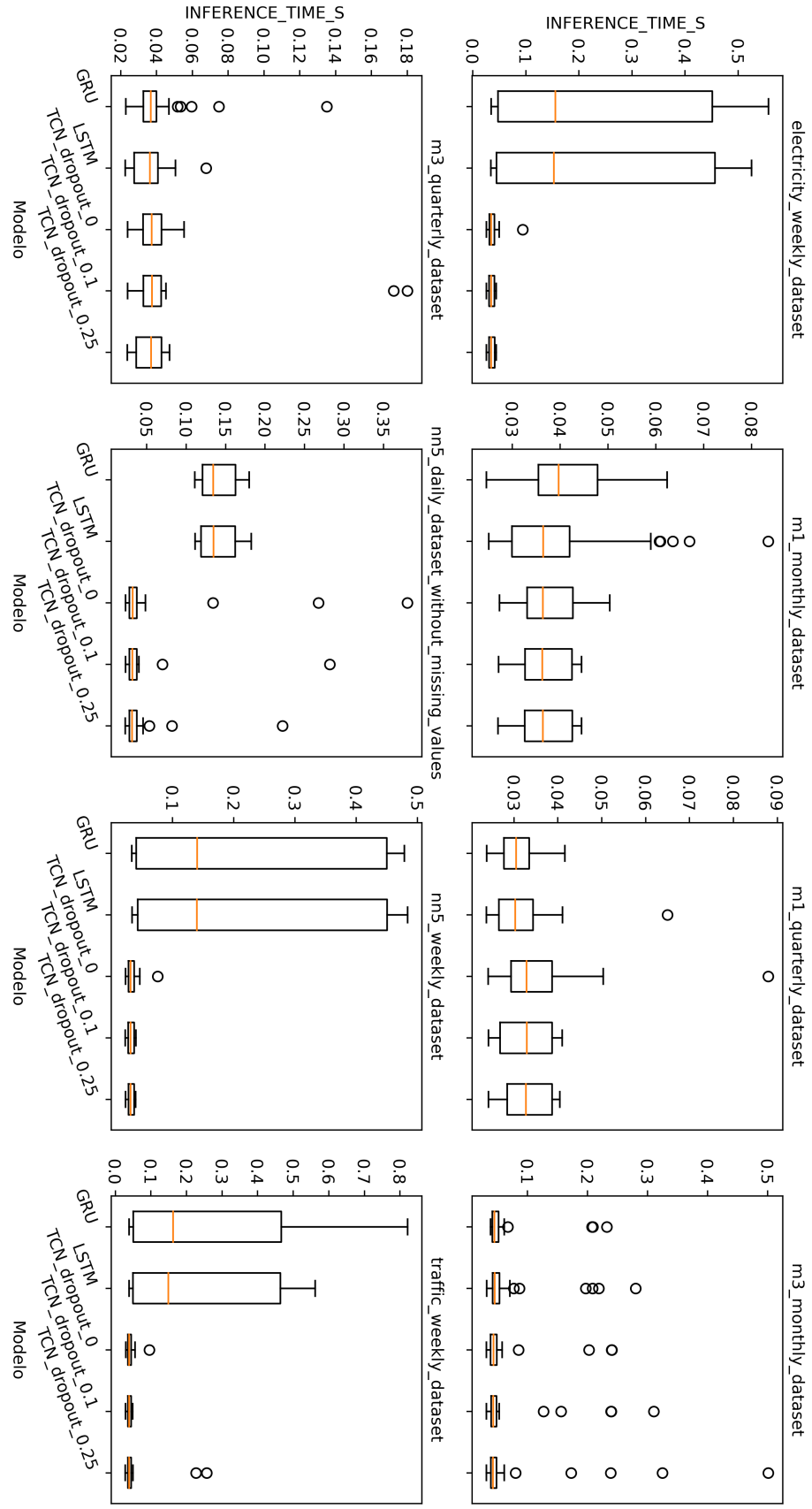

Figura 4.2: Diagramas de cajas y bigotes del tiempo de inferencia (en segundos) para cada modelo y conjunto de datos.

4.2. Análisis de la calidad de las predicciones

Tal y como se comentó en el marco experimental (sección 3.4), para medir la calidad de las predicciones hemos usado tres métricas: RMSE, SMAPE y WAPE. En las figuras 4.3, 4.4 y 4.5 se muestran sus diagramas de bigotes por modelo y conjunto de datos. Al igual que en la sección anterior, podemos encontrar más detalles sobre ellos en las secciones A.3, A.4 y A.5 respectivamente.

Comenzaremos comentando los resultados obtenidos según el RMSE. Observamos que el valor más común de los errores pertenece al orden de 10^6 para todos los modelos en el conjunto de datos *electricity*, cuyos valores se sitúan entre 10^3 y $3 \cdot 10^8$ aproximadamente. Las mejores configuraciones de las redes con celdas GRU y LSTM han logrado alcanzar un error medio de orden 10^5. Sin embargo, las grandes desviaciones presentes en las observaciones tomadas también se dan entre los errores cometidos en esta métrica. En ambos conjuntos *m1* también sucede un fenómeno similar, presentando un valor mediano y medio elevado con una desviación significativa. Esto puede deberse a las características de las series que componen estos conjuntos porque sus series son más heterogéneas según el análisis exploratorio realizado, costando más trabajo a los modelos aproximarlas y aprender sus características relevantes.

Respecto al resto de conjuntos de datos, obtenemos los mejores resultados en *San Francisco traffic* usando TCN, presentando el error medio y desviación más bajos. El uso de *dropout* al 10 % mejora muy levemente el rendimiento del modelo. En los conjuntos `nn5` las LSTM han destacado respecto a las GRU por su baja desviación (alrededor de 0.5 y 2.0 respecto a un 1.14 y 5.7), obteniendo un rendimiento similar y siendo igualas por las TCN. Por último, en el conjunto `m3` con frecuencia mensual no presenta diferencias significativas, aunque las desviaciones de los modelos LSTM son mucho menores que el resto (60.80 respecto a 314.11 de las GRU y aproximadamente 100 respecto a las TCN). Si miramos este mismo conjunto con frecuencia cuatrimestral, predominará la TCN sin *dropout* al tener un error mediano y medio próximo al 980 y una desviación mínima (31.99 respecto a un valor medio de 68.5 del resto de TCN y a desviaciones superiores a 100 por parte de LSTM y GRU).

Si hablamos sobre el WAPE o el SMAPE también ocurre que a medida que el dataset se vuelve menos heterogéneo, las métricas y sus desviaciones son cada vez más bajas. Respecto al SMAPE, nos percatamos de que el conjunto de datos *electricity* no es el más difícil, siendo estos los dos conjuntos correspondientes a *m1*. El valor de la mediana en el primer conjunto mencionado abarca el intervalo $[0{,}169, 0{,}231]$, mucho

mejor que $[0{,}434, 0{,}581]$ para la frecuencia mensual y $[0{,}461, 0{,}572]$ para la cuatrimestral respectivamente. En general, como sabemos que estas métricas tienen en cuenta la escalas de los datos, podemos afirmar que hemos llegado a encontrar configuraciones de nuestros modelos que se adaptan bien a todos los conjuntos de datos usados. El comportamiento del WAPE es similar al del SMAPE para todos los conjuntos, exceptuando *electricity* y los *m1*, disminuyendo el valor de la mediana en todos los casos, excepto en los modelos TCN del primer dataset.

Modelo	Media Medias	Media Std.	Std. Medias	Std. Std.
GRU	0.286	0.113	0.206	0.094
LSTM	0.277	0.065	0.207	**0.067**
TCN_dropout_0	**0.240**	0.070	**0.145**	0.084
TCN_dropout_0.1	0.246	0.070	0.151	0.079
TCN_dropout_0.25	0.256	**0.062**	0.164	0.073

Tabla. 4.1: Media de medias, media de desviaciones, desviación de las medias y desviación de las desviaciones para cada modelo de la métrica SMAPE.

Modelo	Media Medias	Media Std.	Std. Medias	Std. Std.
GRU	0.196	0.123	0.079	0.108
LSTM	**0.168**	0.062	**0.058**	**0.064**
TCN_dropout_0	0.171	0.059	0.067	0.082
TCN_dropout_0.1	0.174	0.055	0.064	0.070
TCN_dropout_0.25	0.177	**0.053**	0.065	0.072

Tabla. 4.2: Media de medias, media de desviaciones, desviación de las medias y desviación de las desviaciones para cada modelo de la métrica WAPE.

Las tablas 4.1 y 4.2 nos aportan una visión general del rendimiento de los modelos, obtenidas al agregar las métricas SMAPE y WAPE de todos los conjuntos de datos. Observamos que no existen diferencias muy significativas, pero sí lo suficientemente relevantes como para afirmar que un modelo funciona mejor que otro. La métrica SMAPE nos indica que la TCN sin *dropout* obtiene una mejor marca que el resto junto con una menor media y desviación de medias, lo que nos informa de que ha conseguido resultados robustos y similares entre todos los dominios evaluados. La diferencia respecto al resto de modelos en desviación no es elevada, excepto en las redes recurrentes con celdas GRU, portadoras de los peores resultados. Si nos fijamos en la métrica WAPE tenemos diferencias aún más insignificantes, siendo ahora las redes con LSTM las que alcanzan los mejores resultados y las GRU los peores con desviaciones mayores respecto a las obtenidas con el SMAPE.

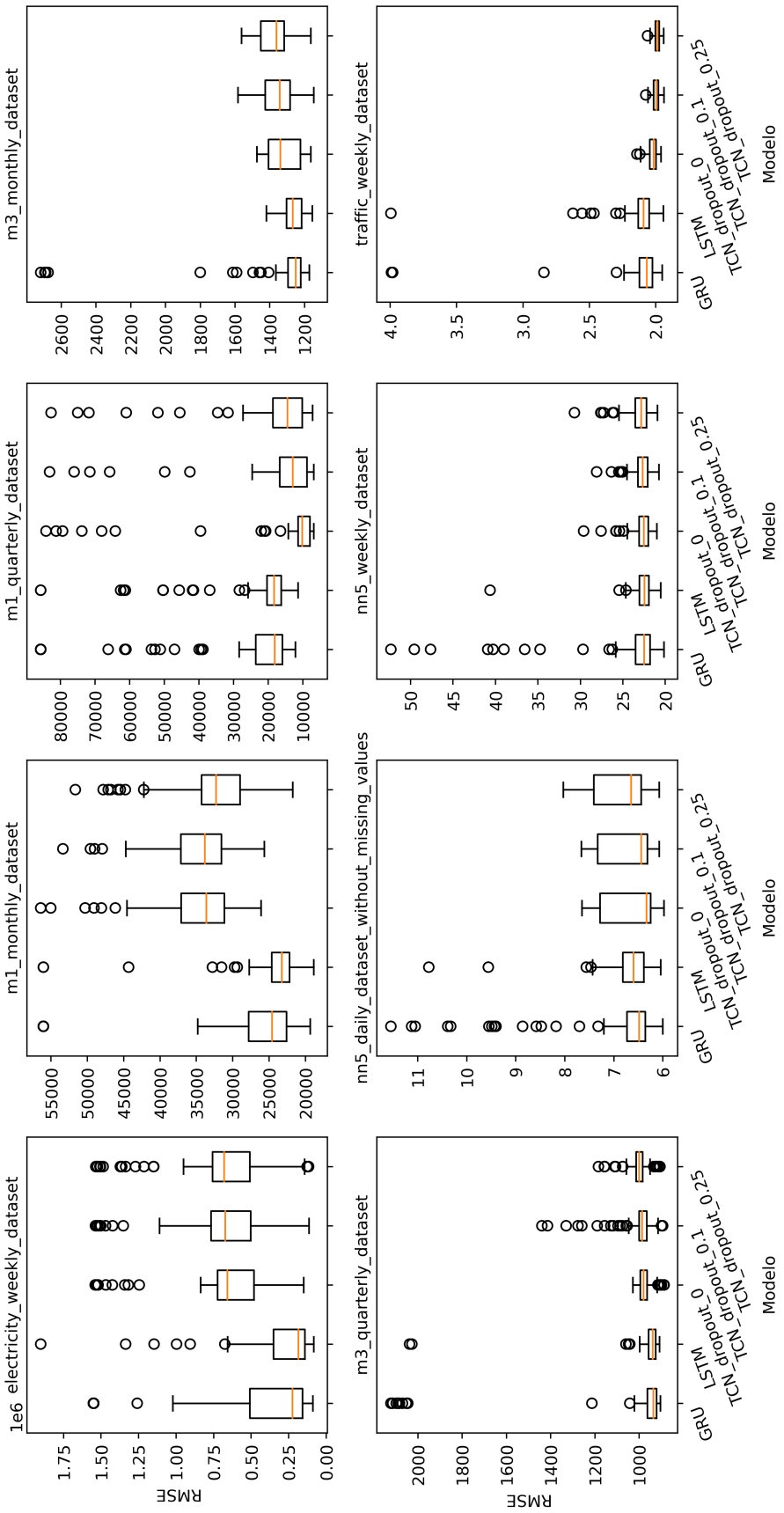

Figura 4.3: Diagramas de cajas y bigotes del RMSE para cada modelo y conjunto de datos.

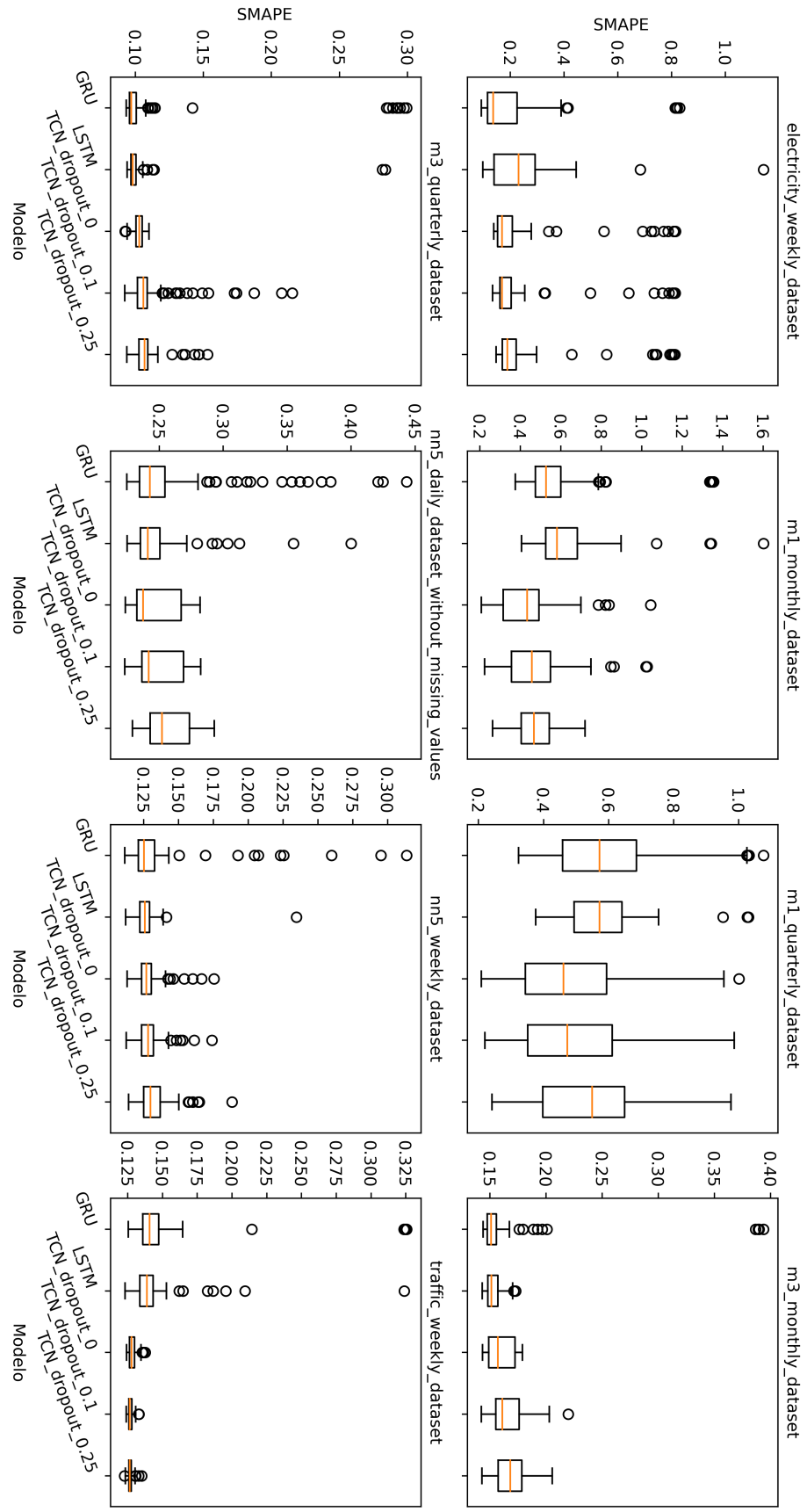

Figura 4.4: Diagramas de cajas y bigotes del SMAPE para cada modelo y conjunto de datos.

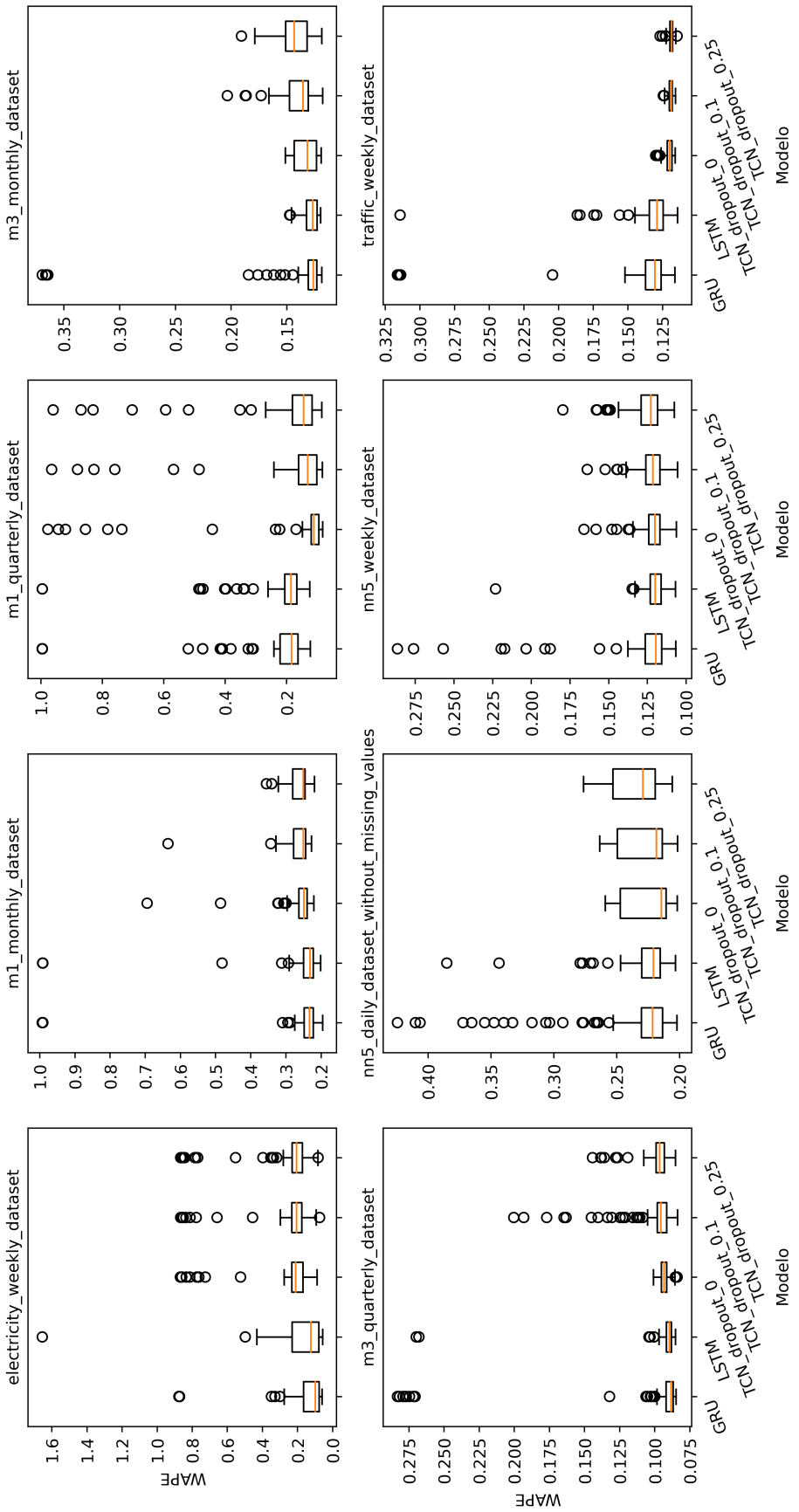

Figura 4.5: Diagramas de cajas y bigotes del WAPE para cada modelo y conjunto de datos.

4.3. Análisis de hiperparámetros

Para finalizar este capítulo, se han escogido las 25 mejores configuraciones de cada conjunto (200 en total) y se han ordenado según la métrica WAPE, ya que nos permite comparar ejecuciones entre diferentes conjuntos y tiene en cuenta las diferentes escalas que puedan poseer nuestras series.

Mostraremos en la tabla 4.3 qué parámetros han sido los preferidos para cada uno de los modelos. Según la frecuencia de aparición en el *ranking* (columna 'Frecuencia' en la tabla) apreciamos que los modelos más seleccionados son las TCN y, curiosamente, las redes con celdas GRU. Si bien en la sección 4.2 hemos confirmado que ellas obtenían los peores resultados generales, tenemos que recordar que sus desviaciones eran mucho más altas que las del resto de modelos, poniendo de manifiesto la existencia de distribuciones de hiperparámetros que funcionan bastante bien y, al mismo tiempo, bastante mal en el mismo conjunto de datos.

Modelos	Frecuencia	Capas	Dim_h	Épocas	LR	BS
TCN_dropout_0	52	4, 1, 2	32∼64, 16	200, 100, 50	10^{-2}, 10^{-3}	16, 32
GRU	51	1, 2, 4	32, 64∼128	50, 200, 100	10^{-3}, 10^{-2}	32, 16
LSTM	40	1, 4, 2	128, 32, 64	200, 50, 100	10^{-3}, 10^{-2}	16, 32
TCN_dropout_0.1	40	4, 1, 2	64, 32, 16	200, 100, 50	10^{-2}, 10^{-3}	16, 32
TCN_dropout_0.25	17	4, 1∼2	64, 16, 32	50, 100, 200	10^{-2}, 10^{-3}	16, 32

Tabla. 4.3: Preferencias de hiperparámetros para cada modelo. Dos valores unidos por una virgulilla (∼) significa que se han seleccionado con la misma frecuencia.

Las redes TCN coinciden en que su número óptimo de capas (en este caso, bloques recurrentes) es 4. Se inclinan por un tamaño oculto de 64 dimensiones y han obtenido sus mejores resultados al entrenarlas durante 200 épocas con un ratio de aprendizaje igual a 0.01. El tamaño de lote preferido es 16, lo que le permite realizar con más frecuencia la optimización de sus pesos. Las TCN con el *dropout* más severo han sido las menos seleccionadas.

Las redes recurrentes han optado por una arquitectura más simple, comúnmente de 1 sola capa, pero con un estado oculto mayor en las LSTM respecto a las GRU. También difieren en el número de épocas y tamaño de lote.

4.4. Ficheros de resultados completos

Para descargar los resultados detallados de la experimentación, haga ***doble clic*** en los *clips* de la tabla 4.4. Los ficheros de la columna RNN contienen los resultados recogidos de las redes recurrentes con celdas LSTM y GRU.

Conjunto de datos	Frecuencia	RNN	TCN
electricity	semanal		
m1	mensual		
m1	cuatrimestral		
m3	mensual		
m3	cuatrimestral		
nn5	diaria		
nn5	semanal		
San Francisco traffic	semanal		

Tabla. 4.4: Resultados detallados de la experimentación por conjunto de datos. Haga **doble *clic*** en el ***clip*** del fichero que desee descargar.

Capítulo 5

CONCLUSIONES

Para finalizar, se expondrán las conclusiones generales derivadas del desarrollo de todo este proyecto. También se comentará el conocimiento que se ha adquirido mediante el su desarrollo y el trabajo futuro que se puede proponer a partir de él.

5.1. Conclusiones

La elaboración de este trabajo ha confirmado la capacidad que poseen los algoritmos de aprendizaje profundo para construir conocimiento de calidad a partir de los datos proporcionados como entrada una y otra vez. Sin embargo, para que ello sea posible es necesario invertir una parte considerable de nuestro tiempo en tres tareas fundamentales: el **análisis** de datos, su **preprocesamiento** y la **optimización** de los hiperparámetros del modelo.

Gran parte de la experimentación realizada en este trabajo se puede ahorrar usando técnicas automáticas dedicadas a encontrar el conjunto de hiperparámetros adecuado para nuestros modelos, sin embargo, se ha decidido proseguir con el enfoque de *fuerza bruta* solamente para intentar estudiar cómo afecta la aplicación de sus diferentes valores a cada una de las arquitecturas, tal y como se demuestra en el capítulo anterior.

Tras analizar los resultados y observar el comportamiento de cada uno de los modelos, podemos afirmar que las redes recurrentes con celdas GRU suelen tardar menos en entrenar, seguidas de las LSTM y por último las TCN (aunque, si nos fijamos en los tiempos mínimos obtenidos, ellas logran ser entrenadas más rápido que los métodos recurrentes). En tiempos de inferencia sucede lo contrario, gracias a su paralelismo intrínseco dado por el operador de convolución que usan.

Respecto a calidad, ocurre un suceso bastante curioso notable gracias al uso de varias métricas. Según el RMSE, las redes recurrentes con celdas LSTM han obtenido el menor error medio en casi todos los conjuntos de datos, seguidas por las TCN y las redes con celdas GRU. Sin embargo, si nos fijamos en el SMAPE y WAPE ambas métricas coinciden en que las TCN han alcanzado una calidad mayor, seguidas de las LSTM. Se atribuye este cambio a las características de las métricas usadas porque el RMSE penaliza duramente errores grandes y **no** tiene en cuenta la escala de las series, a diferencia del WAPE y SMAPE que sí escalan el error. Dicho esto es acertado decir que las dos métricas mencionadas nos han permitido confirmar **la emisión de predicciones** situadas alrededor del rango esperado.

Es necesario mencionar que en conjuntos de datos abarcados en este proyecto, la relación entre el tiempo de entrenamiento de estos modelos y los resultados que se obtienen no es la más adecuada. Como se ilustra en la tabla 4.3 el número de épocas más frecuente es 200, lo que requiere un esfuerzo computacional considerable en comparación a otras técnicas basadas en métodos estadísticos o aprendizaje

automático. Es necesario **no** olvidar la existencia de estas técnicas para adecuarnos a nuestro problema, resolviéndolo garantizando una mínima calidad y **eficientemente**.

5.2. Conocimientos adquiridos

El conocimiento que se ha adquirido desde el inicio hasta el fin del trabajo es considerable. Este trabajo me ha permitido comprender los fundamentos del aprendizaje profundo, así como introducirme en el mundo del análisis de series temporales para posteriormente trabajar en su predicción. Destacar también la destreza conseguida en *PyTorch* y *PyTorch Lightning* al implementar los modelos recurrentes y la TCN, la cual me será útil de cara al futuro.

En el ámbito personal debo mencionar que me ha ayudado a conocerme a mí mismo ante una situación que no llegué a contemplar: la pérdida de toda la experimentación realizada. Supuso un gran retraso en la finalización de este trabajo, y durante dos semanas no pude realizar ninguna ejecución en el clúster debido a un fallo de disco. La coordinación y comunicación con el técnico responsable permitió volver a ejecutar los *scripts* de la experimentación lo antes posible. Debido a este incidente, fue necesario reducir una notable cantidad de las pruebas a realizar (porque, en un principio, también se iban a incluir modelos *transformer*), reorientando así el rumbo del TFM para finalizarlo en la fecha esperada.

5.3. Trabajo futuro

Este trabajo puede evolucionar a un estudio más amplío con la inclusión de dichos modelos *transformers* en él. También, se pueden hacer nuevos experimentos para comprobar si los modelos actuales (de conocimiento **global**, es decir, aprenden a partir de todo el conjunto de datos) emiten predicciones de mayor calidad respecto al uso de modelos **locales** (se entrena un modelo para cada serie, en vez de un dataset). Para ello, será también necesario **encontrar** conjuntos de datos con series **mucho más** largas, de tal manera que podamos percibir el verdadero potencial de este tipo de algoritmos.

Apéndice A

RESULTADOS DETALLADOS DE LA EXPERIMENTACIÓN

A.1. Tiempo de entrenamiento (en segundos)

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	3556.847	1109.093	333.272	**1335.924**	**868.383**
LSTM	3736.302	1137.339	308.629	1350.082	920.387
TCN_dropout_0	5492.973	1295.696	211.609	1610.952	1266.738
TCN_dropout_0.1	5776.532	1361.879	212.783	1685.349	1329.251
TCN_dropout_0.25	5738.764	1354.042	251.518	1669.744	1319.322

Tabla. A.1: Estadísticos del tiempo de entrenamiento para el conjunto de datos *electricity* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	3790.242	1174.896	367.117	1444.325	963.776
LSTM	3898.323	893.702	295.325	**1236.091**	**959.491**
TCN_dropout_0	5742.224	1266.89	204.304	1625.336	1329.866
TCN_dropout_0.1	5954.688	1284.955	207.981	1689.284	1372.563
TCN_dropout_0.25	5944.246	1264.632	210.455	1667.444	1359.684

Tabla. A.2: Estadísticos del tiempo de entrenamiento para el conjunto de datos *m1* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	769.466	249.22	79.692	**300.470**	**195.256**
LSTM	799.026	248.728	80.564	301.923	197.007
TCN_dropout_0	1317.462	301.971	47.353	370.822	300.407
TCN_dropout_0.1	1354.374	298.086	47.873	377.359	310.23
TCN_dropout_0.25	1384.211	301.393	47.909	380.725	316.024

Tabla. A.3: Estadísticos del tiempo de entrenamiento para el conjunto de datos *m1* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	14515.19	4690.412	1451.787	5684.879	**3800.868**
LSTM	14580.845	4688.965	1339.255	**5627.902**	3810.118
TCN_dropout_0	20651.11	4788.467	885.347	5782.664	4542.901
TCN_dropout_0.1	21581.973	4925.439	842.75	6049.095	4753.529
TCN_dropout_0.25	21547.446	4821.825	948.409	5928.784	4597.789

Tabla. A.4: Estadísticos del tiempo de entrenamiento para el conjunto de datos *m3* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	2991.53	975.337	289.474	1160.578	**757.139**
LSTM	3202.086	954.822	288.002	**1153.126**	775.694
TCN_dropout_0	4931.837	1131.168	156.186	1393.634	1158.453
TCN_dropout_0.1	5119.769	1205.923	158.901	1454.635	1194.589
TCN_dropout_0.25	5085.103	1193.574	159.53	1445.129	1196.801

Tabla. A.5: Estadísticos del tiempo de entrenamiento para el conjunto de datos *m3* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	9691.487	3328.154	998.384	3830.754	**2346.609**
LSTM	9574.004	3155.818	1003.458	**3767.413**	2430.535
TCN_dropout_0	15985.079	3684.348	674.355	4530.140	3607.170
TCN_dropout_0.1	16844.848	3794.101	752.7	4757.990	3819.200
TCN_dropout_0.25	16887.515	3820.099	697.243	4760.394	3833.561

Tabla. A.6: Estadísticos del tiempo de entrenamiento para el conjunto de datos *nn5* cuya frecuencia es diaria.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	851.025	270.568	78.489	311.053	196.923
LSTM	866.426	265.119	56.981	295.352	194.968
TCN_dropout_0	791.956	190.35	29.337	**226.291**	**178.963**
TCN_dropout_0.1	817.631	180.761	29.85	227.894	185.872
TCN_dropout_0.25	819.499	185.421	42.341	230.261	184.586

Tabla. A.7: Estadísticos del tiempo de entrenamiento para el conjunto de datos *nn5* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	3181.879	1028.402	297.031	**1192.406**	**767.948**
LSTM	3357.909	1000.955	264.509	1194.388	813.240
TCN_dropout_0	4549.629	1021.74	167.849	1308.558	1049.563
TCN_dropout_0.1	4716.651	1061.803	177.599	1344.609	1081.54
TCN_dropout_0.25	4782.887	1078.819	169.908	1359.729	1090.693

Tabla. A.8: Estadísticos del tiempo de entrenamiento para el conjunto de datos *San Francisco traffic* cuya frecuencia es semanal.

A.2. Tiempo de inferencia (en segundos)

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.557	0.156	0.035	0.222	0.182
LSTM	0.525	0.153	0.034	**0.221**	0.181
TCN_dropout_0	0.095	0.035	0.026	0.036	0.008
TCN_dropout_0.1	0.044	0.034	0.025	0.035	**0.005**
TCN_dropout_0.25	0.044	0.034	0.026	0.035	**0.005**

Tabla. A.9: Estadísticos del tiempo de inferencia para el conjunto de datos *electricity* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.062	0.04	0.025	0.041	0.009
LSTM	0.084	0.036	0.025	0.038	0.01
TCN_dropout_0	0.05	0.036	0.027	0.037	**0.006**
TCN_dropout_0.1	0.044	0.036	0.027	**0.036**	**0.006**
TCN_dropout_0.25	0.044	0.036	0.027	**0.036**	**0.006**

Tabla. A.10: Estadísticos del tiempo de inferencia para el conjunto de datos *m1* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.042	0.03	0.024	**0.031**	**0.004**
LSTM	0.065	0.03	0.024	**0.031**	0.006
TCN_dropout_0	0.088	0.033	0.024	0.034	0.008
TCN_dropout_0.1	0.041	0.033	0.024	0.033	0.006
TCN_dropout_0.25	0.04	0.033	0.024	0.033	0.006

Tabla. A.11: Estadísticos del tiempo de inferencia para el conjunto de datos *m1* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.232	0.045	0.038	0.051	**0.029**
LSTM	0.28	0.045	0.032	0.055	0.036
TCN_dropout_0	0.241	0.043	0.032	**0.048**	0.032
TCN_dropout_0.1	0.311	0.043	0.031	0.051	0.039
TCN_dropout_0.25	0.501	0.042	0.031	0.053	0.056

Tabla. A.12: Estadísticos del tiempo de inferencia para el conjunto de datos *m3* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.135	0.037	0.023	0.037	0.012
LSTM	0.068	0.036	0.023	**0.035**	0.008
TCN_dropout_0	0.055	0.037	0.024	0.037	**0.007**
TCN_dropout_0.1	0.18	0.037	0.024	0.039	0.020
TCN_dropout_0.25	0.047	0.037	0.024	0.036	**0.007**

Tabla. A.13: Estadísticos del tiempo de inferencia para el conjunto de datos *m3* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.18	0.134	0.111	0.140	0.023
LSTM	0.182	0.135	0.111	0.141	0.023
TCN_dropout_0	0.381	0.032	0.023	0.039	0.042
TCN_dropout_0.1	0.282	0.032	0.023	**0.034**	0.025
TCN_dropout_0.25	0.222	0.032	0.023	**0.034**	**0.020**

Tabla. A.14: Estadísticos del tiempo de inferencia para el conjunto de datos *nn5* cuya frecuencia es diaria.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.479	0.14	0.034	0.212	0.179
LSTM	0.484	0.14	0.034	0.214	0.178
TCN_dropout_0	0.076	0.032	0.023	0.033	0.007
TCN_dropout_0.1	0.04	0.032	0.023	**0.031**	**0.005**
TCN_dropout_0.25	0.04	0.031	0.024	0.032	**0.005**

Tabla. A.15: Estadísticos del tiempo de inferencia para el conjunto de datos *nn5* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.822	0.162	0.039	0.237	0.194
LSTM	0.562	0.149	0.039	0.226	0.189
TCN_dropout_0	0.096	0.039	0.029	0.040	0.008
TCN_dropout_0.1	0.048	0.038	0.029	**0.039**	**0.006**
TCN_dropout_0.25	0.258	0.039	0.028	0.043	0.028

Tabla. A.16: Estadísticos del tiempo de inferencia para el conjunto de datos *San Francisco traffic* cuya frecuencia es semanal.

A.3. RMSE

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	1552903.375	224728.07	88774.898	436904.097	454498.489
LSTM	1902938.625	184589.867	82197.461	**287614.428**	**262002.251**
TCN_dropout_0	1540354.5	659443.969	149755.328	670686.818	307009.271
TCN_dropout_0.1	1537779.375	673046.531	114313.547	663362.613	308302.797
TCN_dropout_0.25	1536713.75	680822.75	118308.68	683029.861	327813.409

Tabla. A.17: Estadísticos del RMSE para el conjunto de datos *electricity* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	56084.438	24572.357	19333.807	27405.683	9206.669
LSTM	56082.414	23242.455	18833.555	**24225.724**	**5409.099**
TCN_dropout_0	56432.691	33642.475	26104.584	34761.041	5665.000
TCN_dropout_0.1	53385.281	33843.852	25658.893	34764.894	5517.260
TCN_dropout_0.25	51670.012	32278.225	21750.932	32666.035	5712.208

Tabla. A.18: Estadísticos del RMSE para el conjunto de datos *m1* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	85705.82	18145.895	12109.914	29961.260	24807.925
LSTM	85705.391	18323.837	11344.108	21713.865	13249.888
TCN_dropout_0	84169.242	10153.159	6809.653	**13997.682**	15490.764
TCN_dropout_0.1	83137.648	12891.679	6821.906	15835.914	13194.758
TCN_dropout_0.25	82728.297	14415.681	7187.561	17290.540	**13071.865**

Tabla. A.19: Estadísticos del RMSE para el conjunto de datos *m1* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	2718.963	1251.226	1171.754	1335.165	314.112
LSTM	1420.141	1268.428	1155.853	**1265.023**	**60.791**
TCN_dropout_0	1474.584	1338.922	1164.105	1326.214	91.241
TCN_dropout_0.1	1583.079	1343.13	1146.286	1347.495	100.470
TCN_dropout_0.25	1562.927	1361.611	1163.689	1368.059	105.212

Tabla. A.20: Estadísticos del RMSE para el conjunto de datos *m3* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	2123.083	936.009	903.644	1028.065	302.84
LSTM	2038.916	938.601	907.556	**964.428**	149.982
TCN_dropout_0	1028.706	980.519	887.785	973.804	**31.997**
TCN_dropout_0.1	1440.362	986.994	894.091	1003.729	91.532
TCN_dropout_0.25	1184.106	1000.406	906.044	1001.894	46.091

Tabla. A.21: Estadísticos del RMSE para el conjunto de datos *m3* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	11.55	6.488	6.001	6.887	1.147
LSTM	10.777	6.601	6.045	6.677	0.572
TCN_dropout_0	7.652	6.334	5.98	**6.633**	0.536
TCN_dropout_0.1	7.665	6.444	6.076	6.712	**0.527**
TCN_dropout_0.25	8.028	6.647	6.073	6.863	0.543

Tabla. A.22: Estadísticos del RMSE para el conjunto de datos *nn5* cuya frecuencia es diaria.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	52.343	22.491	20.157	24.109	5.677
LSTM	40.643	22.468	20.534	**22.682**	1.965
TCN_dropout_0	29.627	22.53	20.98	22.704	1.289
TCN_dropout_0.1	28.111	22.651	20.746	22.721	**1.150**
TCN_dropout_0.25	30.71	22.851	20.919	23.111	1.513

Tabla. A.23: Estadísticos del RMSE para el conjunto de datos *nn5* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	3.992	2.067	1.951	2.219	0.512
LSTM	3.997	2.091	1.942	2.117	0.213
TCN_dropout_0	2.144	2.017	1.960	2.025	0.036
TCN_dropout_0.1	2.077	1.995	1.938	1.999	0.026
TCN_dropout_0.25	2.062	1.988	1.941	**1.990**	**0.023**

Tabla. A.24: Estadísticos del RMSE para el conjunto de datos *San Francisco traffic* cuya frecuencia es semanal.

A.4. SMAPE

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.831	0.137	0.093	0.241	0.227
LSTM	1.143	0.231	0.098	0.239	**0.134**
TCN_dropout_0	0.815	0.169	0.138	0.242	0.190
TCN_dropout_0.1	0.814	0.17	0.135	**0.232**	0.171
TCN_dropout_0.25	0.813	0.189	0.148	0.247	0.171

Tabla. A.25: Estadísticos del SMAPE para el conjunto de datos *electricity* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	1.357	0.527	0.376	0.604	0.248
LSTM	1.603	0.581	0.406	0.624	0.178
TCN_dropout_0	1.045	0.434	0.208	**0.436**	0.146
TCN_dropout_0.1	1.028	0.457	0.223	0.471	0.158
TCN_dropout_0.25	0.72	0.467	0.263	0.478	**0.111**

Tabla. A.26: Estadísticos del SMAPE para el conjunto de datos *m1* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	1.077	0.572	0.323	0.616	0.202
LSTM	1.031	0.572	0.376	0.579	**0.119**
TCN_dropout_0	1.002	0.461	0.209	**0.481**	0.175
TCN_dropout_0.1	0.986	0.473	0.219	0.485	0.165
TCN_dropout_0.25	0.977	0.550	0.242	0.533	0.160

Tabla. A.27: Estadísticos del SMAPE para el conjunto de datos *m1* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.394	0.151	0.144	0.165	0.051
LSTM	0.173	0.151	0.143	**0.153**	**0.007**
TCN_dropout_0	0.179	0.157	0.144	0.160	0.011
TCN_dropout_0.1	0.220	0.161	0.143	0.166	0.015
TCN_dropout_0.25	0.206	0.168	0.143	0.168	0.014

Tabla. A.28: Estadísticos del SMAPE para el conjunto de datos *m3* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.3	0.097	0.093	0.113	0.051
LSTM	0.284	0.098	0.094	**0.102**	0.025
TCN_dropout_0	0.11	0.103	0.093	**0.102**	**0.004**
TCN_dropout_0.1	0.215	0.106	0.092	0.111	0.021
TCN_dropout_0.25	0.153	0.107	0.094	0.108	0.010

Tabla. A.29: Estadísticos del SMAPE para el conjunto de datos *m3* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.444	0.243	0.225	0.259	0.045
LSTM	0.4	0.241	0.225	0.247	0.024
TCN_dropout_0	0.282	0.237	0.223	**0.246**	0.018
TCN_dropout_0.1	0.282	0.242	0.223	0.249	**0.017**
TCN_dropout_0.25	0.293	0.252	0.229	0.257	0.018

Tabla. A.30: Estadísticos del SMAPE para el conjunto de datos *nn5* cuya frecuencia es diaria.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.314	0.125	0.111	0.135	0.033
LSTM	0.235	0.126	0.112	**0.127**	0.012
TCN_dropout_0	0.176	0.127	0.113	0.129	**0.010**
TCN_dropout_0.1	0.174	0.128	0.112	0.129	**0.010**
TCN_dropout_0.25	0.188	0.130	0.114	0.132	0.012

Tabla. A.31: Estadísticos del SMAPE para el conjunto de datos *nn5* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.326	0.141	0.125	0.155	0.049
LSTM	0.324	0.139	0.123	0.142	0.022
TCN_dropout_0	0.138	0.128	0.124	0.128	0.003
TCN_dropout_0.1	0.133	0.127	0.124	**0.127**	**0.002**
TCN_dropout_0.25	0.135	0.126	0.123	**0.127**	**0.002**

Tabla. A.32: Estadísticos del SMAPE para el conjunto de datos *San Francisco traffic* cuya frecuencia es semanal.

A.5. WAPE

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.877	0.099	0.060	0.207	0.255
LSTM	1.655	0.124	0.057	**0.170**	**0.172**
TCN_dropout_0	0.869	0.211	0.088	0.261	0.199
TCN_dropout_0.1	0.868	0.206	0.074	0.248	0.181
TCN_dropout_0.25	0.867	0.205	0.083	0.253	0.186

Tabla. A.33: Estadísticos del WAPE para el conjunto de datos *electricity* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.993	0.233	0.196	0.296	0.212
LSTM	0.992	0.231	0.201	0.253	0.107
TCN_dropout_0	0.695	0.249	0.221	**0.259**	0.052
TCN_dropout_0.1	0.636	0.251	0.227	0.264	0.044
TCN_dropout_0.25	0.356	0.251	0.218	0.262	**0.029**

Tabla. A.34: Estadísticos del WAPE para el conjunto de datos *m1* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.996	0.184	0.123	0.310	0.286
LSTM	0.996	0.186	0.125	0.215	0.127
TCN_dropout_0	0.978	0.111	0.083	**0.157**	0.179
TCN_dropout_0.1	0.965	0.132	0.084	0.170	0.152
TCN_dropout_0.25	0.96	0.145	0.086	0.184	**0.150**

Tabla. A.35: Estadísticos del WAPE para el conjunto de datos *m1* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.369	0.126	0.119	0.140	0.051
LSTM	0.148	0.127	0.12	**0.129**	**0.007**
TCN_dropout_0	0.151	0.132	0.119	0.134	0.010
TCN_dropout_0.1	0.204	0.136	0.118	0.140	0.015
TCN_dropout_0.25	0.191	0.144	0.119	0.142	0.014

Tabla. A.36: Estadísticos del WAPE para el conjunto de datos *m3* cuya frecuencia es mensual.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.283	0.088	0.085	0.104	0.050
LSTM	0.27	0.09	0.085	0.094	0.024
TCN_dropout_0	0.101	0.093	0.084	**0.093**	**0.004**
TCN_dropout_0.1	0.2	0.096	0.084	0.101	0.021
TCN_dropout_0.25	0.144	0.097	0.085	0.098	0.010

Tabla. A.37: Estadísticos del WAPE para el conjunto de datos *m3* cuya frecuencia es cuatrimestral.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.425	0.221	0.202	0.238	0.047
LSTM	0.385	0.221	0.203	0.227	0.024
TCN_dropout_0	0.259	0.214	0.202	**0.224**	**0.019**
TCN_dropout_0.1	0.263	0.218	0.201	0.228	**0.019**
TCN_dropout_0.25	0.276	0.229	0.206	0.235	**0.019**

Tabla. A.38: Estadísticos del WAPE para el conjunto de datos *nn5* cuya frecuencia es diaria.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.287	0.12	0.107	0.129	0.031
LSTM	0.223	0.12	0.107	**0.121**	0.011
TCN_dropout_0	0.166	0.12	0.106	0.122	**0.009**
TCN_dropout_0.1	0.164	0.121	0.105	0.122	**0.009**
TCN_dropout_0.25	0.18	0.123	0.108	0.125	0.011

Tabla. A.39: Estadísticos del WAPE para el conjunto de datos *nn5* cuya frecuencia es semanal.

Modelo	Máximo	Mediana	Mínimo	Media	Desv
GRU	0.316	0.13	0.116	0.145	0.049
LSTM	0.314	0.129	0.114	0.133	0.021
TCN_dropout_0	0.13	0.12	0.116	0.120	0.003
TCN_dropout_0.1	0.125	0.119	0.116	**0.119**	**0.002**
TCN_dropout_0.25	0.127	0.118	0.115	**0.119**	**0.002**

Tabla. A.40: Estadísticos del WAPE para el conjunto de datos *San Francisco traffic* cuya frecuencia es semanal.

Bibliografía

Eric Clower. Introduction to the fundamentals of time series data and analysis, Oct 2022. URL https://www.aptech.com/blog/introduction-to-the-fundamentals-of-time-series-data-and-analysis/. Último acceso: 10/11/2023.

Orhan G. Yalçin. 4 intersecting domains that you can easily confuse with artificial intelligence, Dec 2020. URL https://towardsdatascience.com/4-intersecting-domains-that-you-can-easily-confuse-with-artificial-intelligence-2233cb6ad7d1. Último acceso: 10/10/2023.

Howard Hamilton. Overview of the kdd process, Sep 2000. URL https://www2.cs.uregina.ca/~dbd/cs831/notes/kdd/1_kdd.html. Último acceso: 12/10/2023.

S. Chandramouli, S. Dutt, A. Dāśa, and an O'Reilly Media Company Safari. *Machine Learning*. Pearson Education India, 2018. URL https://books.google.es/books?id=R6ZJzQEACAAJ.

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2023. doi: https://doi.org/10.48550/arXiv.2106.11342.

José Cuartas. El concepto de la convolución en gráficos, para comprender las convolutional neural networks (cnn) o redes convolucionadas, Jan 2021. URL https://josecuartas.medium.com/el-concepto-de-la-convoluci%C3%B3n-en-gr%C3%A1ficos-para-comprender-las-convolutional-neural-networks-cnn-519d2eee009c. Último acceso: 18/11/2023.

Harshvardhan GM, Mahendra Kumar Gourisaria, Manjusha Pandey, and Siddharth Swarup Rautaray. A comprehensive survey and analysis of generative models in machine learning. *Computer Science Review*, 38:100285, 2020. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2020.100285. URL https://www.sciencedirect.com/science/article/pii/S1574013720303853.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017. doi: https://doi.org/10.48550/arXiv.1706.03762.

Sigmundo Preissler. Seasonality in python: Additive or multiplicative model?, Nov 2018. URL https://sigmundojr.medium.com/seasonality-in-python-additive-or-multiplicative-model-d4b9cf1f48a7. Último acceso: 03/12/2023.

Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alexander Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. 2016. doi: https://doi.org/10.48550/arXiv.1609.03499.

Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *CoRR*, abs/1803.01271, 2018. doi: https://doi.org/10.48550/arXiv.1803.01271.

Google Trends. Interés de los términos "tensorflow" y "pytorch" en los últimos 5 años., Jan 2024. URL https://trends.google.es/trends/explore?date=today%205-y&q=tensorflow,pytorch&hl=es. Último acceso: 10/01/2024.

Papers With Code. Papers with code - papers with code: Trends, 2024. URL https://paperswithcode.com/trends. Último acceso: 10/01/2024.

StackOverflow. What's the difference between "hidden" and "output" in pytorch lstm?, Jan 2018. URL https://stackoverflow.com/a/48305882. último acceso: 13/01/2024.

PYPL, 2023. URL https://pypl.github.io/PYPL.html. Último acceso: 10/01/2024.

Orhan G. Yalcin. Top 5 deep learning frameworks to watch in 2021 and why tensorflow, Jan 2021. URL https://towardsdatascience.com/top-5-deep-learning-frameworks-to-watch-in-2021-and-why-tensorflow-98d8d6667351. Último acceso: 10/11/2024.

Apache Software Foundation. Apache mxnet, 2024. URL https://attic.apache.org/projects/mxnet.html. Último acceso: 10/01/2024.

Talent.com. Salario medio para científico de datos en España, 2024. URL https://es.talent.com/salary?job=data. Último acceso: 10/01/2024.

Amazon. Instancias p3 de amazon ec2, 2024. URL https://aws.amazon.com/marketplace/pp/prodview-ius6gr3mf5gsm. Último acceso: 25/01/2024.

High-Level Expert Group on Artificial Intelligence. A definition of AI: Main capabilities and scientific disciplines, Apr 2019. URL https://www.aepd.es/sites/default/files/2019-12/ai-definition.pdf. Último acceso: 09/10/2023.

John McCarthy. What is artificial intelligence? Jan 2004. URL https://philpapers.org/rec/MCCWIA.

Abdullah A. Abonamah, Muhammad Usman Tariq, and Samar Shilbayeh. On the commoditization of artificial intelligence. *Frontiers in Psychology*, 12, Sept 2021. ISSN 16641078. doi: 10.3389/fpsyg.2021.696346.

Nick Bostrom. How long before superintelligence? *International Journal of Futures Studies*, 2, 1998.

Keith D. Foote. A brief history of data science, Oct 2021. URL https://www.dataversity.net/brief-history-data-science/. Último acceso: 15/12/2023.

Giuliano Liguori. Data science history and overview. *Data Science History and Overview*, Nov 2020. URL https://www.kdnuggets.com/2020/11/data-science-history-overview.html. Último acceso: 15/12/2023.

Aya Rizk and Ahmed Elragal. Data science: developing theoretical contributions in information systems via text analytics. *Journal of Big Data*, 7, Dec 2020. ISSN 21961115. doi: 10.1186/s40537-019-0280-6.

Yangyong Zhu and Yun Xiong. Defining data science. 2015. doi: https://doi.org/10.48550/arXiv.1501.05039.

Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37, Mar 1996. doi: 10.1609/aimag.v17i3.1230. URL https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1230.

Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–444, May 2015. ISSN 14764687. doi: 10.1038/nature14539.

Beamlab. Deep learning 101 - part 1: History and background, 2017. URL http://beamlab.org/deeplearning/2017/02/23/deep_learning_101_part1.html. Último acceso: 14/10/2023.

Ahmad Abdullah. Machine learning evolution-the story of perceptron, Oct 2020. URL https://towardsdatascience.com/machine-learning-evolution-the-story-of-perceptron-b1de3f180ed6. Último acceso: 14/10/2023.

Pablo Insente. The adaline - theory and implementation of the first neural network trained with gradient descent, 2020. URL https://pabloinsente.github.io/the-adaline. Último acceso:c 25/01/2024.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015. doi: https://doi.org/10.48550/arXiv.1502.01852.

Yuhan Bai. Relu-function and derived function review. *SHS Web of Conferences*, 144: 02006, 2022. doi: 10.1051/shsconf/202214402006.

Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017. doi: 10.1109/ICEngTechnol.2017.8308186.

Afia Zafar, Muhammad Aamir, Nazri Mohd Nawi, Ali Arshad, Saman Riaz, Abdulrahman Alruban, Ashit Kumar Dutta, and Sultan Almotairi. A comparison of pooling methods for convolutional neural networks. *Applied Sciences*, 12(17):8643, Aug 2022. ISSN 2076-3417. doi: 10.3390/app12178643. URL http://dx.doi.org/10.3390/app12178643.

Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019, 2022. doi: 10.1109/TNNLS.2021.3084827.

Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *CoRR*, abs/1912.05911, 2019. doi: 10.48550/arXiv.1912.05911. URL http://arxiv.org/abs/1912.05911.

Timothy P Lillicrap and Adam Santoro. Backpropagation through time and the brain. *Current Opinion in Neurobiology*, 55:82–89, 2019. ISSN 0959-4388. doi: https://doi.org/10.1016/j.conb.2019.01.011. URL https://www.sciencedirect.com/science/article/pii/S0959438818302009. Machine Learning, Big Data, and Neuroscience.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1310–1318, Atlanta, Georgia, USA, Jun 2013. PMLR. URL https://proceedings.mlr.press/v28/pascanu13.html.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. 2014. doi: https://doi.org/10.48550/arXiv.1406.2661.

Panagiotis Antoniadis. Applications of generative models. *Baeldung on Computer Science*, Jun 2023. URL https://www.baeldung.com/cs/applications-of-generative-models. Último acceso: 29/11/2023.

Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. 2021. doi: https://doi.org/10.48550/arXiv.2012.07436.

Robert B Cleveland, William S Cleveland, Jean E McRae, and Irma Terpenning. STL: A seasonal-trend decomposition procedure based on loess. *J. Off. Stat*, 6(1):3–73, 1990.

James Mackinnon. Critical values for cointegration tests. *Long-Run Economic Relationships*, Feb 1990.

Souhaib Ben Taieb, Gianluca Bontempi, Amir F. Atiya, and Antti Sorjamaa. A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition. *Expert Systems with Applications*, 39(8):7067–7083, 2012. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2012.01.039. URL https://www.sciencedirect.com/science/article/pii/S0957417412000528.

Jamal Fattah, Latifa Ezzine, Zineb Aman, Haj El Moussami, and Abdeslam Lachhab. Forecasting of demand using arima model. *International Journal of Engineering Business Management*, 10:1847979018808673, 2018. doi: 10.1177/184797901880867 3. URL https://doi.org/10.1177/1847979018808673.

C. Chatfield. The holt-winters forecasting procedure. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 27(3):264–279, 1978. ISSN 00359254, 14679876. doi: https://doi.org/10.2307/2347162. URL http://www.jstor.org/stable/2347162.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, Dec 1997. doi: 10.1162/neco.1997.9.8.1735.

Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. doi: https://doi.org/10.48550/arXiv.1412.3555. URL http://arxiv.org/abs/1412.3555.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. doi: https:

//doi.org/10.48550/arXiv.1602.07868. URL https://proceedings.neurips.cc/paper_files/paper/2016/file/ed265bc903a5a097f61d3ec064d96d2e-Paper.pdf.

Javier Canales Luna. Top programming languages for data scientists in 2023, Mar 2023. URL https://www.datacamp.com/blog/top-programming-languages-for-data-scientists-in-2022. Último acceso: 10/01/2024.

Rakshitha Godahewa, Christoph Bergmeir, Geoffrey I. Webb, Rob J. Hyndman, and Pablo Montero-Manso. Monash time series forecasting archive. In *Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

Rakshitha Godahewa. TSForecasting. https://github.com/rakshitha123/TSForecasting/blob/master/utils/data_loader.py, 2021.

Francisco Martínez, María Pilar Frías, María Dolores Pérez, and Antonio Jesús Rivera. A methodology for applying k-nearest neighbor to time series forecasting. *Artificial Intelligence Review*, 52(3):2019–2037, 2019. doi: https://doi.org/10.1007/s10462-017-9593-z.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

Hansika Hewamalage, Christoph Bergmeir, and Kasun Bandara. Recurrent neural networks for time series forecasting: Current status and future directions. *International Journal of Forecasting*, 37(1):388–427, 2021. doi: https://doi.org/10.1016/j.ijforecast.2020.06.008DOI(s)linkingtorelatedresources.

PyTorch. Smoothl1loss, 2023. URL https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html. Último acceso: 12/01/2024.

Jens Pohl. Artificial superintelligence: Extinction or nirvana?, 2015.

D. Dickey and Wayne Fuller. Distribution of the estimators for autoregressive time series with a unit root. *JASA. Journal of the American Statistical Association*, 74, Jun 1979. doi: 10.2307/2286348.

William Wei. *Time Series Analysis: Univariate and Multivariate Methods*, volume 33. Jan 1989. ISBN 978-0-201-15911-0. doi: 10.2307/2289741.

Colin Lea, René Vidal, Austin Reiter, and Gregory D. Hager. Temporal convolutional networks: A unified approach to action segmentation. *CoRR*, abs/1608.08242, 2016. doi: https://doi.org/10.48550/arXiv.1608.08242. URL http://arxiv.org/abs/1608.08242.

Pedro Lara-Benítez, Manuel Carranza-García, and José C Riquelme. An experimental review on deep learning architectures for time series forecasting. *International journal of neural systems*, 31(03):2130001, 2021. doi: https://doi.org/10.1142/S0129065721300011.

Bernard Widrow and Michael A. Lehr. Artificial neural networks of the perceptron, madaline, and backpropagation family. 1993. URL https://api.semanticscholar.org/CorpusID:59849930.