Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Creación de un Modelo de Predicción de Ventas Basado en Modelos de Redes Neuronales Recurrentes.

Autor: Víctor Campos Sánchez.

Tutor(a): Raúl Alonso Calvo.

Madrid, Enero 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Creación de un Modelo de Predicción de Ventas Basado en Modelos de Redes Neuronales Recurrentes.
Enero 2025

Autor: Víctor Campos Sánchez

Tutor:
Raúl Alonso Calvo
Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

El presente Trabajo de Fin de Grado (TFG) se centra en el desarrollo e implementación de un modelo de predicción de ventas utilizando redes neuronales recurrentes (RNN, por sus siglas en inglés) como las redes neuronales LSTM. Dichas redes neuronales recurrentes (RNN) son una solución innovadora a la par que práctica capaz de capturar, modelar e inferir patrones temporales complejos presentes en datos como son las ventas. A diferencia de las redes neuronales tradicionales, las RNN tienen la capacidad de recordar información de estados anteriores de la secuencia de datos, lo cual es crucial para la predicción de series temporales como las ventas.

Esto, hace de las redes recurrentes una tecnología idónea para la predicción de ventas. Predicción la cual es una tarea fundamental a la par que critica para las empresas, ya que permite estimar la producción, gestionar inventarios y optimizar los recursos.

Por tanto, este trabajo empieza explorando los conceptos teóricos que rodean a las redes neuronales, desde sus inicios hasta el desarrollo de las redes neuronales recurrentes. Exploración en la que se van exponiendo de forma detallada esos conceptos clave para entender el funcionamiento de las redes neuronales tanto tradicionales como recurrentes. Con ello, se añade una serie de ejemplos prácticos que no solo sirven para que el autor pueda ir conociendo el funcionamiento de las herramientas y los modelos a usar, sino que también se realizan como añadido a la explicación teórica. Pudiendo de esta forma completar dichas explicaciones al experimentar con dichos modelos y observar sus comportamientos.

Finalmente, como resultado de este trabajo, se usarán estos conceptos para buscar de qué forma se pueden usar redes neuronales recurrentes (en concreto redes LSTM) para desarrollar un modelo de predicción de ventas, que pueda ser usado por las empresas como herramienta de estimación de niveles de producción.

Abstract

The present Final Degree Project (FDP) focuses on the development and implementation of a sales prediction model using recurrent neural networks (RNN), specifically Long Short-Term Memory networks (LSTM). These recurrent neural networks (RNN) provide an innovative and practical solution capable of capturing, modelling, and inferring complex temporal patterns found in data such as sales. Unlike traditional neural networks, RNNs could retain information from previous states in a sequence, which is crucial for predicting time-series data like sales.

This makes recurrent networks an ideal technology for sales forecasting. Such forecasting is a fundamental yet critical task for businesses, as it allows them to estimate production, manage inventory, and optimize resources.

Therefore, this work begins by exploring the theoretical concepts surrounding neural networks, from their origins to the development of recurrent neural networks. This exploration presents these key concepts in detail, providing a clear understanding of how both traditional and recurrent neural networks function. Alongside this, a series of practical examples are included, not only to help the author become familiar with the tools and models to be used but also to complement the theoretical explanations. In this way, the explanations are enriched by experimenting with the models and observing their behaviors.

Finally, as the outcome of this project, these concepts will be applied to explore how recurrent neural networks, particularly LSTM networks, can be utilized to develop a sales prediction model. Such a model could serve as a valuable tool for businesses to estimate production levels.

Tabla de contenidos

1 Introducción

Hoy en día, uno de los campos tecnológicos más en tendencia y auge se podría decir que es la Inteligencia Artificial. Su evolución a lo largo de estos últimos años aparentemente ha sido clave para poder dar un nuevo salto y una nueva evolución en lo que respecta a la industria [1]. Pudiendo así ver la Inteligencia Artificial implementada e integrada en diversas áreas como la ofimática, la salud, la banca o en otras muchas más. Esto se debe a que las ventajas y los beneficios que aportan son muy extensos y amplios [2]. Dentro de la Inteligencia Artificial existen también muchas áreas o campos que plantean diferentes tecnologías con diferentes propósitos. Sin embargo, este trabajo se centra en las redes neuronales. En concreto en las Redes Neuronales Recurrentes (RNN). Acto seguido se desarrollará e indagará en los distintos tipos que hay y en diversas características que tienen, además de su aplicabilidad.

Como comienzo, remontándonos al nacimiento o los inicios de las redes neuronales, en el año 1943 los investigadores Warren McCulloch y Walter Pitts buscaban una forma de recrear la sinapsis del cerebro humano en los dispositivos electrónicos. Para ello desarrollaron un modelo informático denominado lógica umbral que se basa en una serie de algoritmos matemáticos [3]. Más adelante, tras ciertos avances, el psicólogo Frank Rosenblatt en 1958 desarrolló el denominado Perceptrón el cual tuvo como objetivo el simular procesos biológicos como el aprendizaje en dispositivos electrónicos. Después como gran descubrimiento y mejora significativa de las redes neuronales Paul John Werbos en 1974 dio las primeras pinceladas en el desarrollo del algoritmo "back-propagation" el cual mejoraba considerablemente el aprendizaje y ajuste de las redes neuronales [4][5][6]. Sin embargo, no fue hasta 1986 que Geoffrey Hinton (actual ganador del Premio Nobel de Física junto con John Hopfield) acompañado de sus compañeros terminaron de darle forma al algoritmo de "back-propagation" [7].

Gracias a estas investigaciones y a muchas otras más. Actualmente se cuenta con una gran variedad de redes neuronales que difieren en ámbito de uso y arquitectura. Sin embargo, se puede decir que todas nacen de los mismos fundamentos. Dentro de dicha variedad de redes neuronales que hay actualmente se puede encontrar las Feedforward Neural Networks – FNN, las Convolutional Neural Networks – CNN (con arquitecturas derivadas como LeNet, AlexNet o VGG, entre otras), las Radial Basis Function Networks – RBF, Generative Networks (con arquitecturas derivadas como Deep Convolutional GAN – DGGAN, CicleGAN o StyleGAN), Transformers… No obstante, las redes neuronales en las que se trabajará son las Redes Neuronales Recurrentes o como se dice en ingles Recurrent Neural Networks – RNN [8][9]. Este tipo de redes neuronales nace como evolución de las redes neuronales tradicionales Feedforward Neural Networks – FNN. Sin embargo, en su arquitectura añaden la cualidad de que ciertos nodos o perceptrones puedan retroalimentarse de sus propias salidas anteriores. Esto le da a la red neuronal la capacidad de memoria y por lo tanto de recordad que es lo que se ha hecho en instantes de tiempo pasados. Por lo tanto, parece ser que las RNN son una herramienta muy útil para lo que es el procesamiento de secuencias temporales. En otras palabras, parecen ser muy útiles para trabajar con datos secuenciales donde es importante ver que valores preceden al valor analizado en un cierto instante de

tiempo, y dependiendo de estos se obtiene una salida u otra. No como las FNN, que trabajan con datos donde cada entrada produce una salida sin mirar lo ocurrido anteriormente y de forma aislada al resto de entradas y salidas. Por ejemplo, se tienen datos como frases, palabras, pronósticos medioambientales... Esto da a ver que las RNN a diferencia de las FNN al procesar datos trabajan con un contexto que afecta a la salida que se pueda obtener [10].

Por otra parte, dentro de las Redes Neuronales Recurrentes – RNN se tienen diferentes arquitecturas derivadas de esta. Por un lado, las Long Short-Term Memory (LSTM) las cuales introducen celdas de memoria para retener información. Una muy buena opción cuando se trabaja con secuencias temporales muy largas como datos. Lo que las hace menos susceptible a problemas como gradiente explosivo o desvanecedor. Por otro lado, están las Gated Recurrent Units (GRU) las cuales son una variante más simplificada de LSTM que consta de menor cantidad de parámetros. Lo cual las hace más eficiente en el proceso de aprendizaje, pero quizás menos efectivas en la resolución. Y finalmente, las Bidirectional RNN (BRNN) las cuales procesan secuencias temporales en ambas direcciones pudiendo así aprender a partir de las primeras o de las últimas partes de la secuencia. Ósea pudiendo aprender del pasado o del futuro de las secuencias (en este presente TFG no se abordarán las redes neuronales GRU ni BRNN) [11].

1.1 Motivación

Actualmente, a diferencia de la mentalidad del pasado, a la forma de innovar de la sociedad se han añadido temas y objetivos nuevos como la eficiencia o la preocupación medioambiental. Esto se puede ver reflejado en movimientos y proyectos actuales como la agenda 2030. Por lo tanto, aspectos como la disminución del consumo de recursos (energía, combustibles fósiles o tiempo), el uso de energías renovables o la implantación de una economía circular, entre otros, han pasado a ser objetivos fundamentales de la mayoría de las organizaciones e instituciones más grandes del mudo. Uno de los factores que afecta de forma directa al malgasto de recursos es la situación de las empresas de tener un gran stock en los almacenes sin salida a venta. Una situación fácil de darse debido a la dificultad en la predicción del comportamiento de los consumidores y a la gran cantidad de variables a tener en cuenta.

Por ello, este trabajo consiste en la investigación sobre el desarrollo de un modelo basado en las Redes Neuronales Recurrentes (RNN) que ayude con la realización de análisis de secuencias temporales como son las ventas producidas en un determinado tiempo. Teniendo como objetivo influir en la disminución del uso de recursos y los desajustes desmedidos de previsión de stock. La forma en la que se podría lograr este objetivo sería gracias a la información proporcionada por las predicciones de ventas realizadas por medio del modelo basado en RNNs. Una información que ayudaría a los vendedores a medir con más exactitud la producción y el stock necesarios para satisfacer la demanda de los compradores. Suponiendo así, un ahorro en materias primas involucradas en la producción de esos productos que se quedan en los almacenes sin salida al mercado.

De esta forma, el objetivó de este trabajo de fin de grado (TFG), como ya se ha mencionado antes, pasa por la implementación de un modelo de RNN eficiente y eficaz, especializado en predicciones de ventas. Un modelo general que pueda servir para multitud de empresas y escenarios. Que pueda ser capaz de adaptarse a las diferentes variedades y diferencias que tienen los diferentes sectores económicos a los que pertenecen las empresas. Un modelo que aproveche la potencia de las redes neuronales con el aliciente de ser capaz de tener en cuenta el contexto de su entorno, ósea memoria, gracias a las cualidades de las sucesiones temporales.

Será tarea de este TFG ver cuál es el mejor modelo dentro de las RNN para trabajar con sucesiones temporales como son las ventas. Quizás sea mejor aplicar una bidireccionalidad para poder analizar datos de un contexto pasado o futuro. O quizás sea necesario aplicar modelos como el LSTM para poder trabajar con sucesiones largas, o incluso modelos como GRU ya que sería posible simplificar el modelo. Todo ello dependerá de la naturaleza de los datos y lo que estos demanden.

Así mimos, varios serán los objetivos a alcanzar en este trabajo para poder obtener el modelo de RNNs deseado. Siendo este lo suficientemente preciso y efectivo a la par que eficiente.

2 Estado del Arte

2.1 Marco Teórico

2.1.1 Redes Neuronales Artificiales RNAs

Las redes neuronales artificiales (RNA), son un campo de la inteligencia artificial del cual sus inicios se remontan a la década de 1940 con el desarrollo de un modelo informático llamado lógica umbral por Warren McCulloch y Walter Pittis [23]. El neurólogo Warren McCulloch y el matemático Walter Pitts, introdujeron lo que fue la primera idea de la neurona artificial en su publicación de 1943 "A Logical Calculus of Ideas Immanent in Nervous Activity" [42]. En este artículo los autores presentaron un modelo en el que las neuronas podrían funcionar gracias a la realización de cálculos complejos mediante lógica proposicional. A partir de ese punto y gracias a numerosos trabajos como el del psicólogo Frank Rosenblatt con su artículo titulado "Perceptron Simulation Experiments" [20], donde desarrolla la idea del Perceptrón, se fue elaborando y perfeccionando una herramienta capaz de mediante ejemplos poder aprender patrones y por tanto simular el aprendizaje por experiencia del ser humano. Esto ha creado un gran interés sobre las RNAs debido a su gran utilidad en predicciones y pronósticos, pudiendo así usarlas en multitud de campos de aplicación de ámbitos económicos, sociales, técnicos... Así mismo, con el paso del tiempo, las RNAs han ido cambiando y pasando a ser cada vez más sofisticadas, con nuevas y mejores prestaciones gracias a avances en algoritmos de aprendizaje más optimizados, nuevos modelos y a un hardware cada vez más mejorado [43].

Las RNA son un modelo general capaz de inferir una gran variedad de funciones que albergan el significado de relación entre un conjunto de sucesos y un resultado, siendo flexibles y poco susceptibles a casos atípicos, ruido o errores en los datos de entrada dado que las redes neuronales artificiales son una aproximación funcional universal. Así mismo, las RNA normalmente son de carácter no lineal debido a las funciones de activación no lineales usadas por lo que son capaces de encontrar relaciones no lineales entre los datos. Relaciones no lineales como la mayoría de las presentes en la naturaleza [23].

Para su funcionamiento, las RNA cuentan con una serie de características principales e indispensables.

- En primer lugar, los pesos, necesarios para ponderar las conexiones de la red (como las conexiones de las entradas) y representar una cierta relevancia o importancia de la conexión.
- En segundo lugar, el término independiente o bias, valor que logra desplazar a la recta o el hiperplano del origen de coordenadas evitando su paso por este punto. El desplazamiento se puede considerar como un peso más que multiplica a una entrada imaginaria no cambiante de valor uno.
- En tercero y último lugar, la función de activación la cual proporciona el resultado de la salida de las neuronas y por ende de la RNA en su

totalidad. Existen multitud de funciones de activación para usar desde funciones muy simples como la función umbral hasta funciones muy sofisticadas. Sin embargo, para resolver problemas complejos que no son linealmente separables, aparte de usar más capas y neuronas es necesario usar funciones de activación no lineales.

No obstante, hay muchos otros elementos presentes en las RNA que son necesarios para su adecuado funcionamiento, como puedan ser hiperparámetros como la tasa de aprendizaje, el número de épocas, el número de capas o el número de nodos por capa.

De esta forma las RNA gozan de una estructura multicapa compuesta de tres capas enlazadas entre ellas. La capa de entrada compuesta por tantos nodos como variables de entrada haya. La capa oculta, la cual está compuesta de una o múltiples capas de uno o múltiples nodos. Y la capa de salida que está compuesta de nodos los cuales aplican la última función de activación a la función general del modelo, siendo así los nodos con los que se obtiene los valores resultado.

Sin embargo, RNA es un término usado para agrupar las distintas arquitecturas de redes neuronales que hay. El objetivo de este trabajo es ver las características y singularidades de las redes neuronales FFNN, RRNN y LSTM. Teniendo también presente que cada una de estas redes neuronales es un avance y mejora de la anterior para resolver problemas de forma óptima que la anterior no puede.

2.1.2 Perceptrón

En 1958 el psicólogo Frank Rosenblatt publicó el artículo “Perceptron Simulation Experiments” [20] donde explica el concepto del perceptrón como un sistema digital capaz de imitar el comportamiento de una neurona biológica, pudiendo simular su mecanismo de procesamiento de información. La estructura del perceptrón básico se basa en un conjunto de variables de entrada donde cada una tiene asignada un peso. Dichas variables son ponderadas por sus pesos y después sumadas de forma conjunta. Acto seguido, se le aplica un sesgo a esta suma y el resultado se pasa por una función umbral que evaluará 0 si no pasa el umbral o 1 si lo pasa. Dicha función umbral es la función de activación la cual puede variar según el propósito (En la figura 1 se puede ver una explicación más ilustrada). Así mismo, para tener un mejor desempeño se le somete al perceptrón a un proceso de aprendizaje supervisado donde se le introduce un conjunto de entradas y se cerciora de que la salida sea la esperada. Durante dicho entrenamiento, el perceptrón ajusta sus pesos según la funcione de error o coste que se use [20].

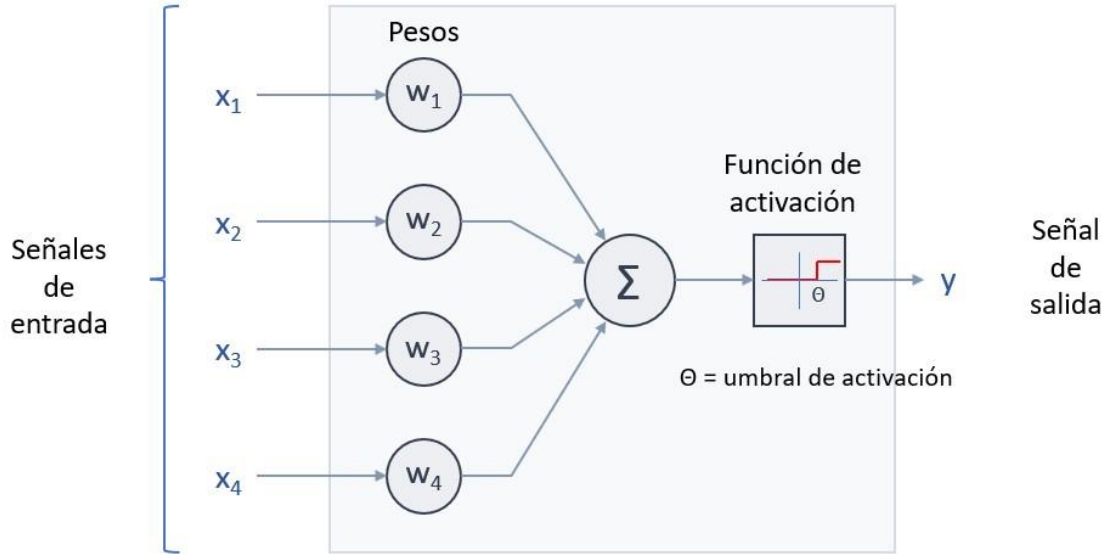


Figura 1. Diagrama ilustrativo de un perceptrón [52].

Rosenblatt en este trabajo demuestra y hace ver que el perceptrón es capaz de clasificar patrones linealmente separables, ósea patrones que se puedan dividir en conjuntos distintos por una línea o por un hiperplano en los casos de conjuntos de variables de entrada de dimensiones mayores. Uno de los ejemplos que usa Rosenblatt para demostrar esto es la realización de tareas de reconocimiento de patrones visuales. Sin embargo, al igual que Rosenblatt destacó esta capacidad de clasificación lineal, también destacó la imposibilidad del perceptrón para poder resolver problemas que no sean linealmente separables como pueda ser el mencionado problema del XOR. No fue hasta la llegada de los perceptrones multicapa y el algoritmo de la retropropagación cuando ya se pudo solucionar esta incógnita [21].

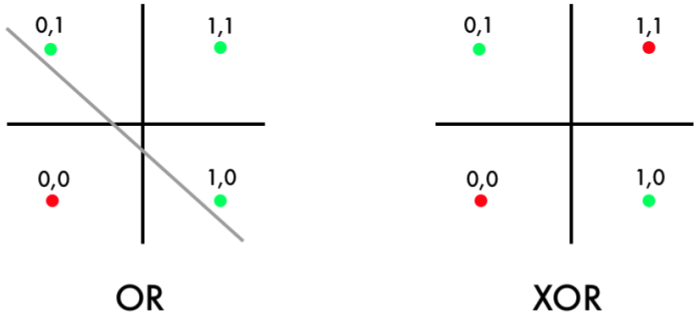


Figura 2. Comparativa entre la posibilidad de separabilidad lineal de la función OR y la no posibilidad de una XOR [46].

De esta forma el trabajo de Rosenblatt fue una de las investigaciones que sentaron las bases en el campo de las redes neuronales dado que modelos inmediatamente posteriores usaron como base la idea del perceptrón. Así mismo estos modelos fueron a su vez base para otros modelos posteriores, y así sucesivamente. Por tanto, se podría decir que la evolución de las redes

neuronales, donde cada modelo busca solventar algunas de las limitaciones de sus antecesores, tiene como punto de partida el perceptrón siendo una de las primeras y más básicas rede neuronal.

2.1.3 Regresión, Interpolación, Sobreajuste y Subajuste

La regresión es un modelo matemático perteneciente a la estadística que se usa para aproximar la relación de dependencia entre una variable dependiente f(x) y un conjunto de variables independientes (x_0, x_1...x_{n-1}, x_n), añadiendo un valor adicional o residuo. Por tanto, mediante la regresión se puede obtener una función que describa la relación que hay entre un conjunto de valores (entrada) y un suceso (salida) de forma aproximada. Teniendo, así como resultado una función que conocidas las variables de entrada puede dar como resultado una salida acorde con la relación natural que intenta simular. Esto se ve presente, en cierta medida, en la forma de funcionar y ser de las redes neuronales. Y eso se debe a que las redes neuronales partieron de la base del modelo de regresión en concreto la regresión lineal.

Otro método matemático relacionado con la regresión es la interpolación. La interpolación es un modelo matemático que se utiliza para estimar valores intermedios de una función o conjunto de datos. Para ello la interpolación, dado un conjunto de datos, obtiene una función que pasa exactamente por todos esos puntos. La diferencia entre un modelo matemático y otro es que mientras que la interpretación busca la función exacta que pase por los puntos, la regresión busca una función que no hace falta que pase por los puntos pero que si tenga una tendencia marcada por estos, que se aproxime.

Ahora, sabiendo esto es lógico pensar que es preferible usar la interpolación debido a su carácter de exactitud y dado que puede obtener una función más exacta y precisa. Sin embargo, esto no es cierto debido a dos razones. Primero este modelo matemático busca una exactitud que no se da en la naturaleza, en esta se pueden hallar errores y rarezas por lo que una red neuronal que intenta ajustarse de forma exacta no es capaz de tolerar bien dichas rarezas y errores naturales. Por lo tanto, se acabaría teniendo una red neuronal la cual aprende de dichos errores y rarezas. Segundo, el proceso de entrenamiento de una red neuronal pasa por comprobar su salida, derivada de ciertas entradas, y comprobar su acierto según los valores reales o esperados que le corresponden a esas entradas. Cunado a una red neuronal se le entrena con una cierta base de datos esta red debe de tener una cierta flexibilidad a datos nuevos dado que ningún set de datos abarca todos los datos posibles. Utilizando la interpolación esto sería un problema dado que ante la aparición de datos nuevos es posible que la red neuronal no pudiera aprender de ellos, dado que con la interpolación se sacrifica la generalidad por un ajuste más exacto.

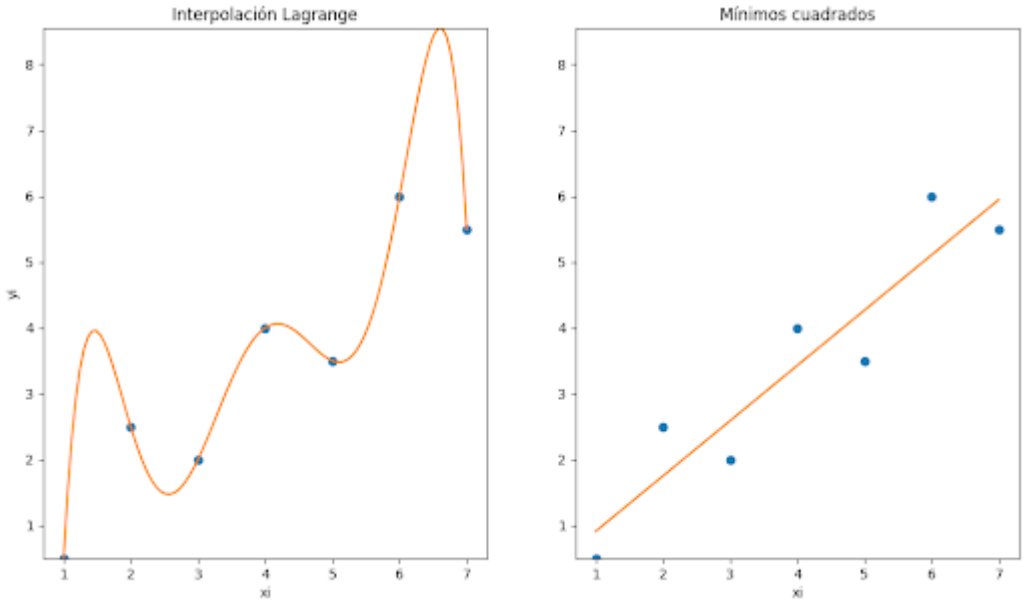


Figura 3. Comparación entre interpolación y regresión.

No obstante, hay investigaciones que han demostrado que la teoría clásica del aprendizaje automático, en la que se afirma que aumentar la capacidad de un modelo más allá de cierto punto da como resultado un sobreajuste, puede estar equivocada para modelos complejos y altamente parametrizado como las redes profundas. Si bien la interpolación del modelo es un problema de sobreajuste, estas investigaciones han observado que, tras aplicarle una complejidad aún mayor, sobrepasando un punto de interpolación, se puede lograr tener un modelo extremamente ajustado y aun así generalizado que tolere bien nuevos datos [22]. Para demostrar esto, introducen un nuevo concepto llamado "curva de doble descenso" (figura 4). La curva de doble descenso describe como el error del modelo puede disminuir incluso después de pasar el punto de interpolación, donde la capacidad del modelo es tan alta que puede ajustarse de forma precisa y exacta a los datos. Por tanto, se desarrolla una nueva teoría moderna para la práctica moderna, donde los modelos pueden alcanzar una mayor precisión sin sacrificar la generalidad. Demostrando así que en estos casos la interpolación no es necesariamente perjudicial [22].

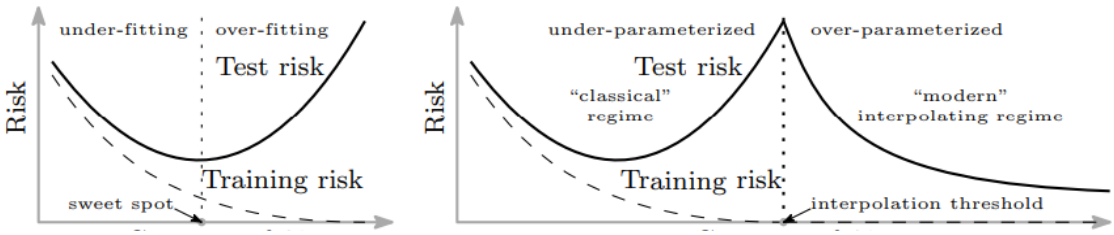


Figura 4. Ilustración de la diferencia entre la curva de error de la teoría clásica y la moderna, obtenida de [22].

Volviendo a la regresión, desde los inicios de las redes neuronales, mismamente con el perceptrón del trabajo de Frank Rosenblatt de 1958, se usó la regresión

lineal la cual se caracteriza por el uso de funciones lineales [20]. Más adelante, con modelos posteriores, se pasó al uso de la regresión logística la cual es parecida a la lineal, pero con la diferencia de que añade una capa más usando una función no lineal en el proceso. Lo cual le permite resolver problemas que la regresión lineal no pude como la clasificación binaria.

En esta regresión logística es en lo que se basa las redes neuronales actuales, pero su enfoque va mucho más allá de cualquier regresión logística simple. La diferencia de las redes actuales con respecto a la regresión, como modelo matemático, reside en el uso de capas ocultas, entre otros aspectos, lo que les da la capacidad de capturar relaciones mucho más complejas. Mientras que una regresión logística realiza el ajuste sobre una función no lineal, una red neuronal aplica un ajuste a una función compuesta de funciones lineales y no lineales anidadas entre sí. Esto le brinda un carácter más general siendo una aproximación funcional universal que puede aproximar con precisión a una amplia gama de funciones para, a su vez, aproximarse a una amplia gama de datos, pudiendo abordar con ello una amplia variedad de problemas. Es más, si de una red neuronal solo se usara la capa de entrada y salida, dependiendo de que función se use si lineal o no, la función del modelo correspondería a una regresión lineal o logística [23].

RED NEURONA	$y=\sigma(W(L)\cdot\sigma(W(L-1)\cdot\ldots\sigma(W(1)\cdot X+b(1))+\cdots+b(L-1))+b(L))$
REGRESIÓN LINEAL	$y=w1\ x1\ +w2\ x2\ +\cdots+wn\ xn\ +b$
REGRESIÓN LOGÍSTICA	$y=\sigma(w1\ x1\ +w2\ x2\ +\cdots+wn\ xn\ +b)$

Tabla 1. Comparativa matemática entre una regresión lineal, logística y una red neuronal.

2.1.4 Funciones de Activación

Las funciones de activación son una pieza clave en el desarrollo de redes neuronales pudiendo dar mejores prestaciones y comportamientos más beneficiosos si se usan las opciones correctas. A lo largo del desarrollo de las redes neuronales, desde inicios hasta la actualidad, se han ido usado una gran variedad de funciones de activación. Por ello, ante la gran cantidad de estas, al usarlas hay que saber cuál es la opción adecuada para cada caso.

En los comienzos, con las primeras redes neuronales se usaron funciones de activación lineales lo que daba resultados muy pobres, pudiendo resolver solo problemas de clasificación linealmente separables. Por lo tanto, problemas como el famoso XOR o la clasificación múltiple no eran posibles de resolver con el uso de funciones de activación lineales [20]. Así mismo el uso de más capas

no soluciona nada dado que el conjunto de funciones lineales con el uso de operadores lineales da como resultado una función lineal. Para resolver estos problemas además del uso de más capas y nodos por capa, se necesita el uso de funciones no lineales. Estas a diferencia de las funciones lineales tienen la característica de aplicar curvaturas en la función definida por el modelo de red neuronal, abandonando de esta forma la linealidad.

Muchas son las funciones no lineales que se han ido hallando y que se pueden usar, están la Sigmoide, la Tangente Hiperbolica (Tanh), la ReLU, la Leaky ReLU, la Softmax, la Swish... Sin embago, hay ciertas funciones que pueden dar mejores prestaciones que otras. Se ha demostrado que la función ReLU, por ejemplo, es más efectiva que las funciones Tanh o Sigmoide, llegando a comprobar que puede tardar mucho menos en el entrenamiento y llegar a porcentajes de ajuste o precisión mayores. Así mismo, más tarde en 2017 se halló una nueva función, la función Swish, la cual puede dar mejores resultados que la ReLU en cuanto al entrenamiento de redes neuronales [24].

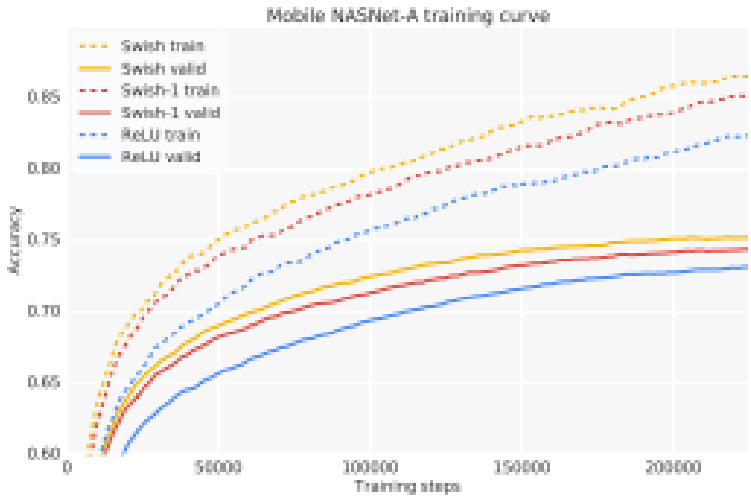
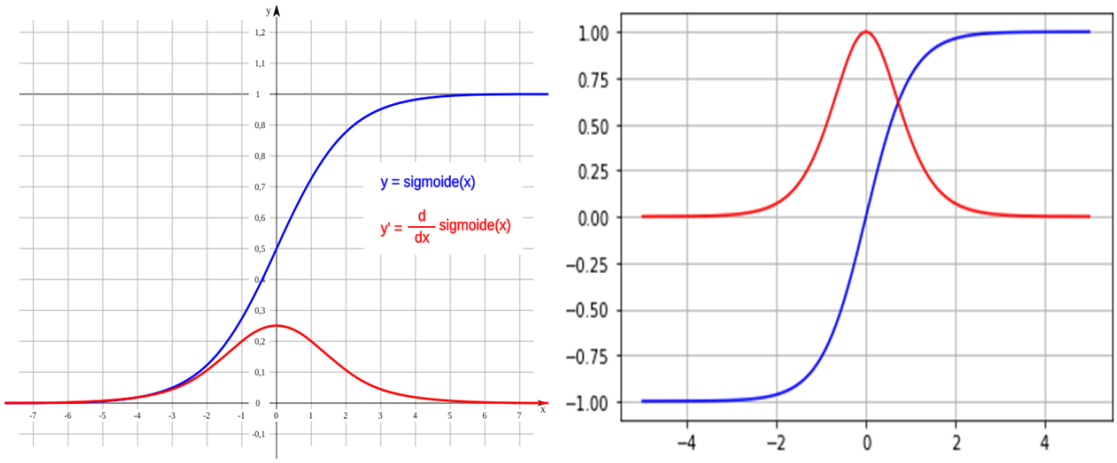


Figura 5. Curva comparativa de entrenamiento de Mobiles NASNet-A en ImageNet, obtenido de [24].

Esta diferencia de prestaciones se da debido a ciertas características que tienen las diferentes funciones. Una de ellas sería el coste de cálculo o computo, siendo un aspecto importante que influye directamente al rendimiento del entrenamiento de las redes neuronales. El costo computacional en cuanto al cálculo de la salida en funciones como la sigmoide o la tangente hiperbólica es significativamente mayor que otras funciones como la ReLU. Por tanto, una red neuronal que usa funciones como la sigmoide o parecidas puede presentar mayores costes de cómputo y tiempo. Provocando consigo un desempeño más pesado en las fases de entrenamiento y posiblemente también en la fase de testeo y producción.

Otra característica sería la susceptibilidad hacia el problema del gradiente desvanecedor. Este ocurre debido a las características de las derivadas de las diferentes funciones de activación si se compara la derivada de la función

sigmoide frente a la derivada de la función tangente hiperbólica, como se puede ver en la figura 6, se ve que la función sigmoide puede alcanzar valores mucho menores que la función Tanh. Esta característica en las funciones de activación puede repercutir en el desarrollo de la retropropagación debido a que aquellos valores de entrada de la función sigmoide reciben una importancia o relevancia en el ajuste y la predicción menor. Lo que conlleva unas variaciones en los pesos muy pequeña y por tanto provoca un peor aprendizaje o incluso la falta de este.



Gráfica de la función sigmoide y su derivada

Gráfica de la función tangente hiperbólica y su derivada

Figura 6. Comparación entre derivada de la función Sigmoide y la Tangente Hiperbólica.

No obstante, la función ReLU también da mejores prestaciones en este aspecto, llegando a tener una mayor velocidad de convergencia y mitigar el problema del gradiente desvanecedor que sufren las funciones Sigmoide y Tanh. Sin embargo, como su derivada o gradiente es siempre 0 con valores negativos, esta función sufre del problema de neuronas muertas [25].

A estos problemas y características se añaden muchos más como el hecho de que en comparación con la función ReLU que para valores negativos da como resultado 0 y a valores positivos el mismo valor (rango [0,inf)). Las funciones Tanh y Sigmoide son funciones acotadas (rangos (-1, 1) y (0, 1)). Por lo que sufren de perdida de información dado que ante cambios entre valores muy grandes o pequeños el resultado no se nota apenas con estas funciones. De esta forma, según se va avanzando se van descubriendo nuevas funciones que solventan problemas de funciones pasadas pero que pueden llevar con ellas problemas nuevos.

2.1.4.1 Funciones de Activación y sus Características

La función Sigmoide, responde con un valor comprendido en el intervalo (0,1) ante cualquier valor de entrada, por lo cual la hace idónea para su aplicación como función de salida en problemas de clasificación binaria de datos. Esta función, además, es derivable en todos los puntos y tiene cualidades que aportan ventajas como la facilidad de derivación, lo que la hace perfecta para

usarla en algoritmos de aprendizaje automático. Sin embargo, como se ha mencionado no hay diferencia notable entre las salidas de entradas de valores altos o entre las salidas de entradas de valores bajos y por tanto su derivada en estos puntos obtiene unos valores muy bajos, lo que puede provocar problemas de desvanecimiento del gradiente y resultar en una tasa lenta de aprendizaje de la red neuronal.

La función Tangente hiperbólica responde con un valor comprendido en el intervalo (-1,1) ante cualquier valor de entrada. En valores cercanos a cero la derivada obtiene valores mayores con respecto a la función sigmoide, lo que resulta en una mayor velocidad de aprendizaje. Además, la función al estar centrada en el cero le permite a la red aprender valores tanto positivos como negativos. Sin embargo, esta función sufre de los mismos problemas que la función sigmoide con respecto a la susceptibilidad al desvanecimiento del gradiente y a la complejidad de su cálculo.

La función ReLU (Función Unidad lineal rectificada) devuelve como salida el mismo valor que la entrada si esta es mayor que cero y si no cero. Una de sus ventajas es la simplicidad de su derivada, dado que para valores de entrada menores que cero su valor es 0 y para valores mayores que cero es 1. Teniendo de esta forma un aprendizaje constante y rápido. Sin embargo, aunque resuelva los problemas de desvanecimiento del gradiente y costes de calculo que tienen las dos funciones anteriores. Debido a ese valor 0 en la derivada de todo valor menor que 0 puede darse la aparición de neuronas “muertas”. Neuronas que dejan de ser consideradas en el aprendizaje, dejando de influir en este.

La función de activación Swish usa la función sigmoide, pero resolviendo o mitigando algunos de los problemas de las funciones sigmoide, ReLU y Leaky ReLU. Entre estos problemas está la inestabilidad en el aprendizaje provocada por la función de activación. Sin embargo, su complejidad de cálculo la hace más lenta para ser calculada con respeto a las anteriores funciones de activación.

Hay muchas otras funciones, sin embargo, estas son las más relevantes dentro del diseño de redes neuronales y merece la pena definirlas.

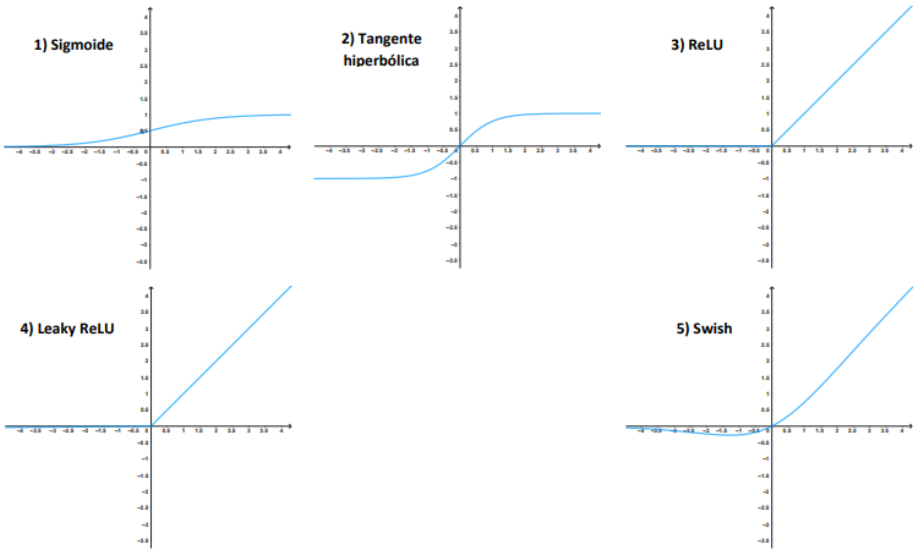


Figura 7. Graficas de las funciones de activación Sigmode, Tangente Hiperbólica, ReLU, Leaky ReLU y Swish.

2.1.5 Redes Neuronales FFNN

Las redes neuronales FFNN fueron uno de los primeros modelos de redes neuronales multicapa desarrollados, siendo la base de muchos de los modelos que actualmente se usan. Dichas redes neuronales, cuentan con al menos una capa de entrada, otra oculta y una de salida, teniendo en cada una un cierto número de neuronas. Esta característica de multicapa es lo que hace posible pasar de un modelo basado en regresión logística o lineal a un modelo más generalista capaz de aproximarse a una gran variedad de funciones [23].

La forma de trabajar de estas redes neuronales es por un flujo secuencial y directo. Las FFNN van aplicando el proceso de suma de entradas ponderadas con sus respectivos pesos, suma del sesgo y aplicación de la función de activación, de forma progresiva capa por capa hasta llegar a la capa de salida. Esto tiene como resultado una función, que define el modelo, compuesta por funciones lineales o no lineales anidadas que toman como entrada las variables de la capa de entrada o el resultado de las neuronas, y por lo tanto funciones de activación, de la capa anterior. Dicha función global del modelo da como resultado un valor final que es fruto de este proceso denominado propagación hacia delante (Fordward Propagation en inglés). Dicho valor final puede haber estado más o menos acertado, pero desde un inicio es difícil que la red neuronal consiga resultados precisos. Por ello, es necesario que se pase por dos fases, la fase de entrenamiento y la de testeo. Entrenamiento para ajustar la red y dar valores precisos en siguientes resultados y testeo para comprobar que la red da esos resultados precisos, y por tanto comprobar su precisión.

En la fase de entrenamiento es preciso tener un conjunto de datos que contenga las variables de entrada y la salida esperada. La forma de operar es la siguiente, se realiza la propagación hacia delante de forma iterativa recogiendo las salidas obtenidas y comparándolas con el valor esperado. Con esta información se

obtiene la función de coste que define en que grado ha estado acertado el modelo. Una vez hecho esto, se realiza el proceso de retropropagación (Backpropagation en inglés) para ajustar los pesos y conseguir que el modelo sea más preciso.

Después de realizar la retropropagación varias veces, se pasa a la fase de testing, donde se realiza el mismo procedimiento que en la fase de entrenamiento solo que sin la parte de la retropropagación. Dado que en esta fase lo que se busca es ver como de precisos son los resultados del modelo, obteniendo así un porcentaje que lo refleje.

2.1.5.1 Back Propagation

Retropropagación o back propagation en igles, es un método de estimación de gradiente usado durante el entrenamiento de redes neuronales artificiales para calcular las actualizaciones de parámetros como los pesos y los bias de las redes neuronales. Dicho método, se basa en el mecanismo del descenso del gradiente de la función de coste, teniendo como objetivo encontrar un punto mínimo de valor cercano o igual a 0.

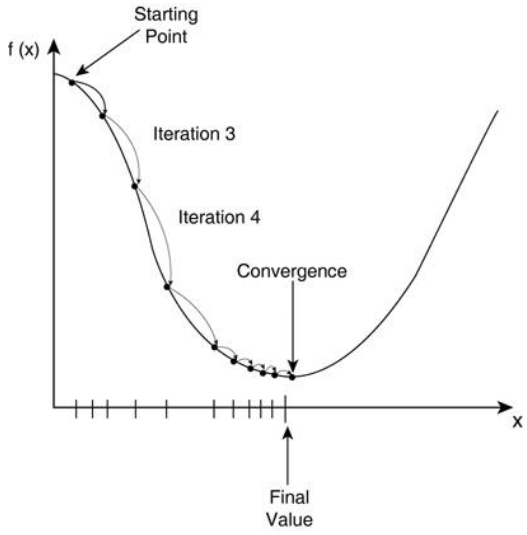


Figura 8. Imagen explicativa del proceso del descenso del gradiente [47].

En otras palabras, teniendo la función de coste que refleja como de equivocado o acertado está siendo el modelo con respecto a los valores de los pesos y bias que se tienen. La retropropagación propaga el error desde la capa de salida, pasando por las diferentes capas ocultas, hasta llegar a la capa de entrada. Propagación que a su paso por las distintas capas se va calculando la derivada de la función de coste con respecto a los pesos y bias de cada capa para así poder actualizar sus valore y aprender. Para la realización de dicha derivada se tiene que aplicar un principio llamado la regla de la cadena (1) [26].

$$\frac{\partial C0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}} \quad (1)$$

Desglosando esta fórmula (1), se ve que la derivada parcial de la función de coste (en este caso el error cuadrático medio) con respecto a los pesos de la capa L, es igual a una sucesión compuesta por la derivada de la transformación afín (z) de la capa L con respecto a los pesos de la capa L, la derivada de la función de activación (en este caso la sigmoide) con respecto a la transformación afín de la capa L y la derivada de la función de coste con respecto a la función de activación de la capa L [26].

$$z^{(L)} = w^{(L)} * a^{(L-1)} + b^{(L)} \quad (2)$$

$$a^{(L)} = \sigma\left(z^{(L)}\right) \quad (3)$$

$$C0 = \left(a^{(L)} - y\right)^2 \quad (4)$$

Como se puede ver en la figura 9, la función de coste (4) toma como entrada la salida de la función de activación de la capa de salida (3). Posteriormente, dicha función de activación toma como entrada el resultado de la transformación afín (2) de las salidas de las neuronas de la capa anterior multiplicadas por sus respectivos pesos y el añadido del bias. Y de esta forma, se va realizando este proceso sucesivamente dado que los valores que son multiplicados por los pesos de las neuronas de la capa L son el resultado de la función de activación de las neuronas de la capa anterior (L-1). Por tanto, se puede notar una relación en cadena entre la función de coste y los pesos de las distintas capas [27].

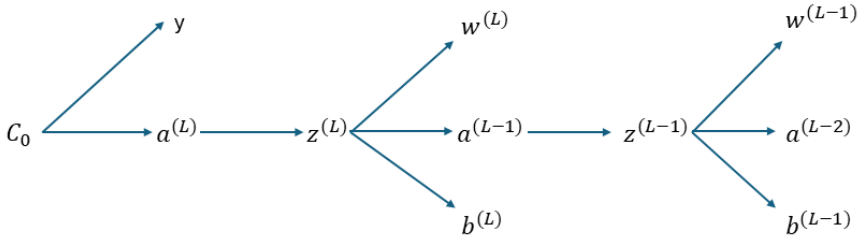


Figura 9. Explicación grafica de la dependencia entre funciones.

Sin embargo, las redes neuronales operan de forma matricial dado que los pesos y los bias de las capas conforman matrices. Por tanto, las fórmulas cambiarían un poco [28].

Una vez sacado los valores de la derivada respecto a los pesos y bias, estos se aplican en la función de actualización donde está presente un hiperparámetro llamado tasa de aprendizaje.

$$w_{ji} = w_{ji} - \alpha * f'(wij) \tag{5}$$

Este hiperparámetro especifica a qué velocidad a de aprender la red neuronal. Su definición es crucial ya que un valor muy alto significa un aprendizaje rápido, pero puede ocurrir que realice cambios de valor muy bruscos provocando que se pase los puntos mínimos (absoluto o locales) constantemente sin poder llegar a ellos y sin poder estabilizarse. No obstante, un valor muy pequeño significa un aprendizaje lento y por ende que tarde mucho en llegar a los puntos mínimos absoluto o los locales [26][27].

Ahora, este proceso no deja de hacer lo mencionado anteriormente que es descender el gradiente por la función de coste. El descenso del gradiente es un método el cual tiene un mejor resultado cuando la función de coste es una función convexa sin presencia de mínimos locales. Esto se debe a que la bajada del gradiente puede llegar a ser más rápida al tener una pendiente más pronunciada a medida que el gradiente se aleje más del mínimo absoluto. Así mismo, la ausencia de mínimos locales hace que al descubrimiento un punto óptimo, que sería el mínimo absoluto, se sepa con certeza que es el mejor valor candidato. Esto hace que funciones de error como la función de error cuadrático medio tengan peor rendimiento que funciones como la función de coste logarítmica en problemas de regresión logística (es un caso en donde se puede usar la función de error logarítmica).

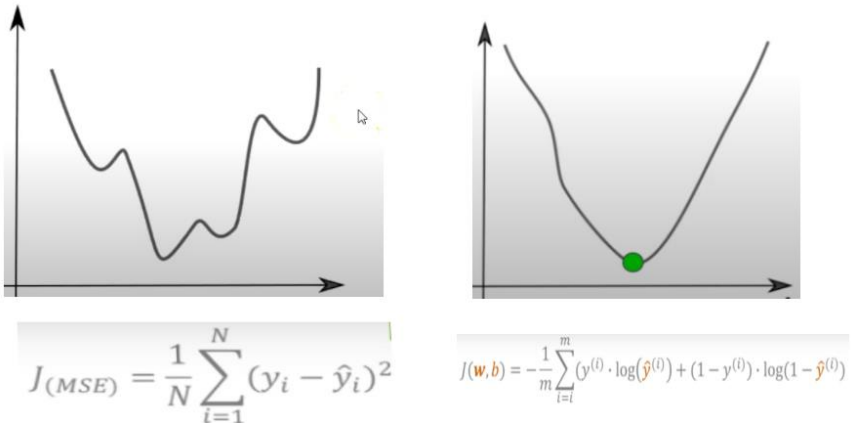


Figura 10. Graficas comparativas de la función de error cuadrático medio y función de coste logarítmica en la regresión logística.

En términos generales y más abstractos, este proceso de retropropagación intenta evaluar cuanto puede afectar el cambio de valor de un peso o un bias al comportamiento de un modelo. Por ello, se podría decir que por la arquitectura o forma de las FFNN los valores de peso y bias de los nodos de capas más cercanas a la salida producen un mayor cambio que los de nodos más cercanos a la capa de entrada. Esto se debe a que los pesos o bias de capas tempranas están influidos por numerosas funciones no lineales mientras que los pesos de capas más cercanas a la capa de salida no se ven tan afectados. Por tanto, a la hora de realizar la derivada de la función de coste con respecto al peso o bias, funciones no lineales como la sigmoide o la tangente hiperbólica van atenuando el valor y por ende la importancia de ese parámetro, dado que las derivadas de estas funciones alcanzan valores muy pequeños (figura 6).

Es por ello por lo que, con FFNNs profundas se sufre de problemas de gradiente desvanecedor dado que los pesos de capas tempranas al no producir casi cambios en el modelo, en la retropropagación estos pesos apenas cambian de valor, lo que provoca un aprendizaje pobre en estas capas. Por otra parte, modelos como las RNN sufren de otro problema llamado gradiente explosivo, el cual ocurre cuando un peso o bias produce un cambio muy significativo en el modelo de red neuronal. Cambio que provoca que en el aprendizaje la red ajuste sus valores de pesos o bias de forma tan brusca que el modelo no sea capaz de estabilizarse y por ende de aprender [26].

Finalmente, el algoritmo de retropropagación se queda obsoleto con ciertos modelos posteriores a las FFNN como son las RNN y sufre de ciertas limitaciones que son solventadas por modelos más avanzados y complejos [29][30].

2.1.5.2 Hiperparámetros

Para un correcto funcionamiento de las redes neuronales hay que tener en cuenta muchos condicionantes. Algunos de los condicionantes más importantes son los considerados hiperparámetros. Dentro de ellos se pueden encontrar ejemplos como el número de capas, el número de nodos por cada capa, la velocidad de aprendizaje, los valores iniciales de pesos y bias o la elección del número de épocas y el tamaño de las muestras para el entrenamiento de la red neuronal. Saber cómo manejarlos puede repercutir positivamente en el rendimiento del modelo de red neuronal que se está usando.

Con respecto al número de capas y nodos, el uso de pocas capas o pocos nodos puede afectar a la red neuronal provocando un subajuste (underfiting en inglés). Subajuste generando por la incapacidad de aprendizaje provocada por la falta de libertad o curvatura en la función que define el modelo para ajustarse a la relación no lineal [31]. Por otra parte, el uso de más capas o nodos de los necesarios provoca un sobreajuste (oberfiting en inglés) debido al exceso de complejidad añadida a la función que define el modelo. Un exceso que puede causar problemas de interpolación, conllevar un gasto de tiempo y procesamiento innecesario, y el acabar teniendo un modelo poco generalista además de poco flexible a nuevos datos [32]. Sin embargo, como se explica en

el punto 2.1.3 “Regresión, Interpolación, Sobreajuste y Subajuste”, recientes artículos hacen ver que esta interpolación y exceso de parametrización de las redes neuronales llegado a un punto no significa forzosamente un problema de sobreajuste [22].

El funcionamiento de las RNA se puede interpretar como una similitud a la forma en que funciona el cerebro humano. El cerebro humano ante un estímulo busca en primer lugar los patrones más simples para después, al entenderlos o aprender de ellos, poder averiguar los patrones más complejos. Pues la misma filosofía de funcionamiento se podría observar en las redes neuronales. El hecho de usar capas proporciona esa escalada en el entendimiento de patrones simples a patrones más complejos. Si se usan pocas capas ese salto de complejidades se vería limitado y quizás haya patrones los cuales el modelo no pueda ver y menos aprender de ellos. Por su contraparte, si se usan más capas de las necesarias el modelo puede que, de la complejidad y la sobreparametrización que albergue, aprenda tan bien de los patrones observados que no pueda llegar a aprender de otros posteriores y diferentes. Mientras tanto, el número de nodos por capa podría afectar en la capacidad de captura de patrones de las RNAs. El uso de pocos nodos pude provocar una deficiencia en la captura de patrones, mientras que el uso de muchos puede derivar en una falta de abstracción durante el aprendizaje.

En cuanto a la elección del tamaño de la muestra para el proceso de entrenamiento, ósea el número de entradas del set de datos que se tienen que usar para cada vez que se tenga que realizar el proceso de retropropagación. Si se hace valor por valor en comparación de por lotes la función de coste es más abrupta produciéndose muchos cambios dispares entre puntos. Sin embargo, con el uso de lotes la función de coste es más suave dado que tiene en cuenta el error promedio de todo el lote, lo que a su vez previene de problemas con mínimos locales. No obstante, usar una muestra tan grande como el conjunto de todos los datos de entrenamiento puede provocar que la función de coste sea muy tenue con lo que la diferencia de un punto a otro sería poco perceptible y la función de error sería poco descriptiva. Por lo tanto, es necesario ver cuál es la magnitud adecuada que se beneficie de la suavidad de usar lotes sin provocar la pérdida de información [33]. Por otra parte, en cuanto a la elección del número de épocas, ósea el número de veces que se usa el conjunto de datos de entrenamiento entero. Si se usa un número pequeño de épocas el modelo no llega a aprender completamente provocando un subajuste, mientras que, si se usa un número de épocas grande, mayor al necesario, el modelo puede sufrir de problemas de sobreajuste perdiendo generalidad y tolerando menos los datos nuevos (aspectos de la tasa de aprendizaje se ven en el punto *2.1.5.1 "Backpropagation*)[34].

Por último, la elección del valor inicial de los pesos y los bias de la red neuronal no es determinante, sin embargo, puede afectar el rendimiento de está dando una mayor velocidad de ajuste o convergencia si se inicializan de forma correcta. Puede haber muchas formas de inicializar los pesos y bias de una red neurona ponerlos todos a 0, a 1, a un valor constante, valores aleatorios, aleatorios acotados… No obstante, métodos como el uso de la distribución normal o el método LeCun podrían dar ese plus en cuanto a velocidad de ajuste. Tan es así,

que por defecto frameworks como Keras usan el método Glorot (parecido a LeCun) para iniciar los pesos y bias [35].

2.1.6 Redes Neuronales RNN

Las redes neuronales RNN solucionan el problema de las redes FFNN con respecto al tratamiento de series temporales. Las redes neuronales FFNN a la hora de operar con sus datos de entrada siguen un flujo secuencial hacia delante por las capas hasta llegar a la capa de salida y obtener el resulta. Sin embargo, las FFNN no son capaces de guardar relación alguna entre el dato procesado actualmente y los datos anteriores. Hay un tratamiento individual y aislado por cada uno de los datos que van entrando a la red. Por ello, las redes FFNN no son una buena solución para tratar datos como las secuencias temporales como puedan ser frases u oraciones [36][37].

Las series temporales se diferencian de los datos individuales tratados en una FFNN por su carácter espacio-temporal. El orden importa en el sentido de que un valor depende del anterior y condiciona al siguiente [37]. Por tanto, en una secuencia como una frase no es lo mismo tener un artículo y un adjetivo como valores previos que tener un sujeto y un verbo. El valor a esperar en el instante de tiempo actual es diferente en ambos casos. Además, el valor que acompañe a estos dos afectará al valor o los valores siguientes.

De esta forma, se puede ver que las FFNN no son capaces de capturar ese orden o carácter espacio-temporal con su tratamiento individual de los datos. Para poder capturar ese orden, se desarrollaron las redes RNN (Recurent Neural Networks). Las RNN al igual que las FFNN usan múltiples capas enlazadas unas con otras. Sin embargo, debido a que el comportamiento de las RNN para un valor esta influenciado por elementos o entradas anteriores [36]. Los nodos de las RNN cuentan con un bucle que guarda la información de salidas previas de este nodo, convirtiéndola en información de entrada. Una entrada llamada “memory cell” que aporta la persistencia de información brindándole la capacidad de memoria al modelo. Se puede ver en la figura 11 como la salida se retroalimenta en la entra [37][38].

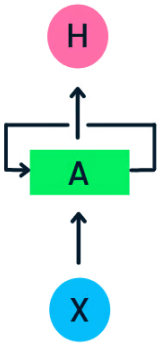


Figura 11. Ilustración de la estructura de una neurona recurrente [48].

Por tanto, en la entrada de la neurona se tienen las variables de entrada x con su respectiva matriz de pesos W_x y la salida del instante anterior $h_{(t-1)}$ con la suya W_h. Lo que da como resultado una la función de salida de esta neurona que sería entrada para instantes posteriores que adquiere la forma de la ecuación (6) (se usa la función sigmoide como función de activación).

$$h_t = \sigma\left(W_h h_{(t-1)} + W_x x_t + b_h\right) \tag{6}$$

Las secuencias temporales están divididas por instancias de tiempo (timesteps en inglés). Por tanto, la neurona recurrente en un instante de tiempo recibe su entrada y a su vez la salida de su instante de tiempo anterior $h_{(t-1)}$. Salida que ha sido calculada a base de las entradas en el instante de tiempo anterior y de la salida del instante de tiempo $h_{(t-2)}$. De esta forma, la operabilidad de las neuronas recurrentes se puede ver bajo una línea temporal como se muestra en la figura 12 [37][38][39].

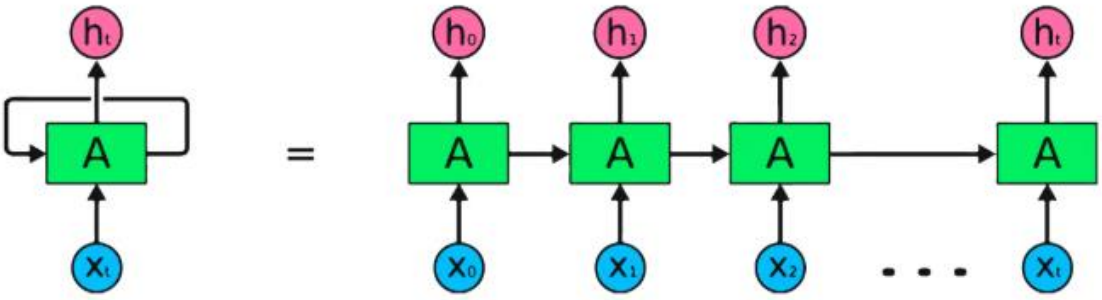


Figura 12. Ilustración del carácter temporal de la operabilidad con sucesiones temporales de una RNN [48].

Los estados o valores de los diferentes teimesteps están referenciados bajo la nomenclatura x_t donde t hace alusión al instante temporal.

Para el entrenamiento y ajuste de pesos de redes RNN no se usa el método de la retropropagación convencional que se usa con las redes FFNN puesto que tiene un rendimiento muy pobre para este tipo de redes neuronales. Por ello, se usa el método de propagación hacia atrás a través del tiempo o BPTT (BackPropagation Throyght Time en inglés). La forma de trabajar de este método es expandiendo la red a través del tiempo, desplegando la RNN por cada instante de tiempo y una vez estirado completamente realizar la retropropagación tradicional [40]. Este estiramiento, hace ver la repetitiva dependencia o relación de un nodo con respecto a los nodos de la anterior capa con sus respectivos pesos y la memory cell y su peso. Repetida aparición o relación que provoca problemas como el desvanecimiento o la explosión del gradiente. Si se hace la derivada de la función de coste respecto a este parámetro que se repite mucho y esta resulta ser de un valor pequeño ($0 < x < 1$) entonces su continua multiplicación va atenuando el valor resultado y por ende su relevancia o

importancia en el modelo, lo que provoca un ajuste pobre o inexistente en esos pesos. Mientras que si el valor es alto (x << -1 || x >> 1) su continua multiplicación con el mismo incrementa su valor exponencialmente provocando una falta de estabilización en el ajuste.

Estos problemas se ven afectados en gran medida por la función de activación usada. Si un peso se ve repetidamente afectado por una función no lineal como la sigmoide o la tangente hiperbólica, estas pueden provocar una atenuación de su valor durante la retropropagación. Mientras que, si se usan funciones como ReLU, al no tener un límite para valores positivos, esta puede provocar el crecimiento desmedido del valor de los pesos y bias. No obstante, una solución sencilla y eficaz para el problema del gradiente explosivo podría ser el establecimiento de un umbral que limite cual es el valor máximo que puede alcanzar un peso o bias. Por otro lado, el problema del gradiente desvanecedor no parece ser tan fácil de solucionar ya que muestra ciertas complicaciones [40].

Por otro lado, se podría decir que los pesos de entradas tempranas dado a su exposición a funciones no lineales pueden ser puntos vulnerables al problema de desvanecimiento del gradiente. Por consiguiente, los pesos de entradas más próximas a la capa de salida pueden ser los más susceptibles a problemas de gradiente explosivo, sobre todo si se inician sus pesos con valores lejanos a 0.

Finalmente, para cerrar esta sección, las RNN se pueden clasificar según el número de entradas y salidas que tiene:

- Varias a varias: Teniendo una secuencia de datos como entradas se obtienen una secuencia de datos de salida. Este tipo a su vez se puede dividir en dos subcategorías dependiendo de si la entrada y la salida están sincronizas o no. En el caso de estar sincronizadas se aplica a problemas como la clasificación de videos, mientras que si no se suele emplear par traducción de idiomas.
- Una a varias: Teniendo una sola entrada de datos se produce una secuencia de datos como salida. Su uso se emplea en campos como el NGL (Natural Lenguage Generation) o la subtitulación de imágenes.
- Varias a una: Teniendo una secuencia de datos como entrada se produce un vector como salida. Tiene gran uno en la clasificación de sentimientos o análisis de opiniones.

2.1.7 Redes Neuronales LSTM

Como se ha visto antes, las redes neuronales RNN sufren de una desventaja muy significativa debido al uso del algoritmo BPTT (Backpropagation Through Time en inglés) para su entrenamiento. El BPTT no deja de ser una iteración más sobre el algoritmo de Backpropagation. Por tanto, el BPTT no solo hereda sus problemas si no que los exacerba más debido a la naturaleza de los problemas que abordan las RNNs.

Esta exacerbación se ve provocada por ese desglose en el tiempo de la red neuronal durante la retropropagación. Desglose que se asemeja a la apertura de un acordeón. Sin extender los acordeones se ve la red neuronal para solo un instante, pero en el momento que se estira el acordeón se ven los diferentes valores que han ido siendo la retroalimentación de cada neurona. A esto hay que añadirle que cada valor de retroalimentación va precedido por una serie de valores de capas anteriores que también tienen sus acordeones … Esto no solo provoca que las RNN sean más propensas a problemas de gradiente desvanecedor y gradiente explosivo, por los motivos mencionados en el anterior apartado, sino que también hace que la velocidad de convergencia en las RNN sea lenta cuando existen estos problemas de gradiente. Además, estas desventajas son más notorias y graves a medida que la serie temporal de entrada sea más larga. Provocando que las redes necesiten mucho más tiempo aun para aprender o que incluso dejen de hacerlo, provocando que la red no memorice y que no pueda reconocer patrones a largo plazo.

Por este motivo se desarrollaron nuevas soluciones, y en este caso nuevos modelos de redes neuronales que solventan estos inconvenientes. Ejemplos de ello son las redes neuronales LSTM o GRU (este TFG solo aborda los conceptos detrás de las redes neuronales LSTM).

Las redes neuronales LSTM Memoria a Largo-Corto plazo (Long-Short Term Memory, en inlges) fueron desarrolladas en 1997 por Hochreiter y Schmidhuber [41]. Creando así un modelo que soluciona las limitaciones de la RNN con secuencias temporales largas al poder proveer a este modelo de memoria a largo plazo para recordar patrones y características entre timesteps alejados. Para poder llegar a esta solución Hochreiter y Schmidhuber focalizaron su esfuerzo en ver una forma de apaciguar o controlar los problemas de desvanecimiento y explosión del gradiente, sobre todo el desvancimiento. Esfuerzo que sirvió para que Hochreiter y Schmidhuber terminaran desarrollando las celdas de memoria dejando atrás todo concepto de neuronas tradicionales. De esta forma, con el uso de dichas celdas, se consigue almacenar datos en un periodo de corto o largo plazo pudiendo añadir nueva información o eliminar la antigua según su importancia y relevancia. Siendo las puertas que contiene la celda de memoria las que, junto con la intervención de los pesos, evalúan que se añade, que se deja y que se elimina de dicha memoria [45].

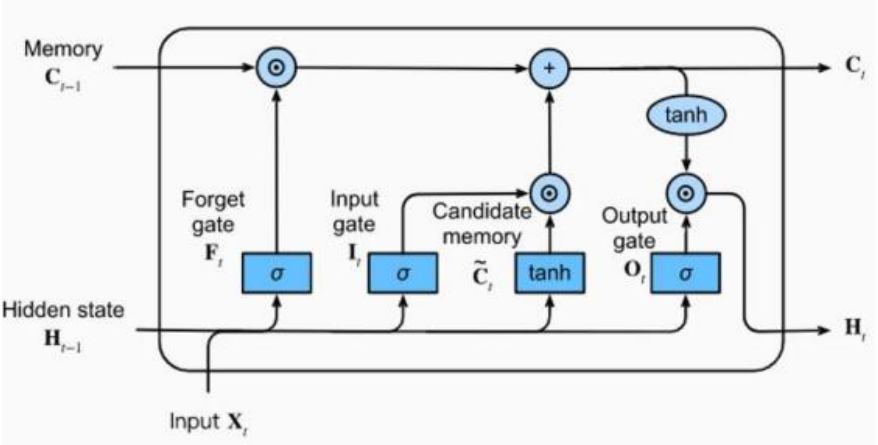


Figura 13. Arquitectura interna de una Celda de memoria de una LSTM con sus respectivas puertas [49].

Las celdas de memoria abordan el problema de memoria a largo plazo gracias a su Estado de Celda (Cell State) C_{t-1}, esta es la entrada y salida que sirve de retroalimentación de la celda de memoria junto con el Estado Oculto (Hidden State). La diferencia entre una y otra reside en que mientras que una se encarga de la memoria a largo plazo la otra se encarga de la memoria a corto plazo y de la salida final. Así mismo, las celdas de memoria LSTM trabajan con el uso de puertas. Dichas puertas son diferentes redes neuronales FNN que al operar junto con operadores vectoriales sacan como salida el estado oculto y el estado de la celda (cell state) de dicho intente de tiempo (timestep). De esta forma se podrían ver dichas puertas como operadores para obtener las salidas [44][49].

Dichas puertas que se usan reciben los nombres de la Puerta de Olvido (Foget Gate), la Puerta de Entrada (Input Gate), la Memoria Candidata (Candidate Memory) y Puerta de Salida (Output Gate). Entre estas puertas se pueden diferenciar dos tipos, las puertas selectoras y las puertas candidatas.

Las puertas selectoras como la Puerta de Olvido, la Puerta de Entrada o la Puerta de Salida comparten el mismo mecanismo de operación (7), pero se usan para propósitos distintos. De inicio, cogen la entrada y el estado oculto del instante de tiempo anterior, concatenándolos en una misma matriz. Dicha matriz se multiplica con unos pesos y se pasa por una función sigmoide. Siendo así el mismo proceso que el de una red neuronal FNN con la capa de entrada y un nodo de salida con función de activación sigmoide [44][49].

$$\sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{7}$$

La razón por la que se usa esta función sigmoide es por la normalización que realiza, buscando el darle al vector selector valores iguales o muy próximos a 0 o 1. De esta forma el uso de este vector selector junto con la multiplicación vector punto a punto puede provocar como resultado el enmascaramiento de los valores a olvidar si en ese campo tiene un valor 0 o puede darle paso si tiene en ese campo el valor 1. Por tanto, en las diferentes pertas se tiene un enfoque distinto:

- Puerta de Olvido: Se busca ver que campos deben tener el valor 0 o próximo en el vector selector para así enmascarar los valores del Estado de la Celda que no se consideren relevantes.

- Puerta de Entrada: Se busca ver que campos deben tener el valor 1 o próximo en el vector selector para ver que valores añadir de la concatenación entre el Estado Oculto y la entrada al Estado de la Celda.

- Puerta de Salida: Mira todos los valores del vector selector, tanto 0s como 1s, puesto que es el filtro que indica que valores del Estado de la Celda usar para generar el Estado Oculto.

Por otra parte, está la puerta candidata o Memoria Candidata. Dicha memoria candidata usa un mecanismo parecido al del resto de puertas con el uso de una red neuronal FNN, pero con la diferencia de que en vez de usar la función sigmoide usa la función tangente hiperbólica (8).

$$\tanh(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{8}$$

El uso de esta función se debe a la necesidad de la normalización también, pero se usa la tangente hiperbólica en vez de la sigmoide debido a que el rango de valores estaría entre –1 y 1 en vez de 0 y 1. Haciendo que este añadido de valores negativos al rango de valores facilite el aprendizaje de patrones más complejos al permitir direcciones opuestas.

De esta forma el uso de las funciones tangente hiperbólica y sigmoide con su característica común de normalización son el mecanismo perfecto para controlar el desvanecimiento y explosión del gradiente.

En resumen, las celdas LSTM tienen como objetivo obtener dos salidas. Por un lado, el Estado de la Celda y por otro el Estado Oculto. Por la parte del Estado de la Celda primero se realiza una multiplicación vectorial punto a punto entre el Estado de la Celda del instante de tiempo anterior junto con el resultado de la Puerta de Olvido [49]. Acto seguido se realiza una suma entre el resultado de la operación anterior más el resultado de la multiplicación vectorial punto a punto entre la Puerta de Entrada y la Memoria Candidata. Teniendo como resultado esta función que define a la salida del Estado de la Celda. Por el otro lado, para el Estado Oculto la operación que se hace es, teniendo en cuenta el resultado actualizado del Estado de la Celda, ósea el resultado de la función anterior, a este se le aplica la función tangente hiperbólica para normalizar el resultado y se le realiza una multiplicación vectorial punto a punto junto con el resultado de la Puerta de Salida [44].

Asimismo, el uso de una única capa de celdas de memoria LSTM es suficiente para trabajar con secuencias temporales elementales y básicas como la clasificación biandria de oraciones o la predicción de la continuación de oraciones cortas. Sin embargo, a medida que se complican los problemas se hace más compleja la red. En el caso de problemas como traducción automática (donde es imprescindible entender el contexto) o el procesamiento de series temporales extensas como una sinfonía, es útil el uso de redes neuronales LSTM profundas. Estas son numerosas capas de celdas LSTM interconectadas entre ellas, la salida de las celdas de capas anteriores se conecta a la entrada de celdas posteriores. De esta manera, como con las redes profundas FNN, las capas tempranas capturan los patrones más simples y pequeños mientras que las capas más próximas a la salida capturan los patrones más complejos y abstractos usando los patrones simples captados anteriormente.

2.2 Tecnologías Usadas.

Dentro del marco de la Inteligencia Artificial, debido al gran impacto social que está teniendo actualmente este campo, la cantidad de herramientas, frameworks y librerías disponibles para su desarrollo es ampliamente extenso. Lo que deja un gran abanico de opciones que puede suponer en los desarrolladores el tener que tomar una decisión sobre cuál de ellas les conviene más usar en sus proyectos.

Por la rama del desarrollo de redes neuronales lo que reina son las librerías. Siendo estas muy variadas y contando con multitud de opciones. Sin embargo, en este presente trabajo se hace uso del lenguaje de programación Python con el añadido de las siguientes librerías.

Para empezar, y como pieza central, se usa Pytorch. Pytorch es una librería o marco de trabajo de código abierto desarrollada por Facebook AI Research (FAIR) para su uso en el lenguaje Python. Ofreciendo de esta forma una API de fácil uso. Esta facilidad de uso la consiguen gracias a su capacidad de abstracción sobre los modelos matemáticos que envuelven a las Redes Neuronales en cuanto a su diseño y proceso de entrenamiento. Entre sus características se encuentran los tensores. El tensor es la estructura de datos abstractos que alberga datos o conjuntos de datos como la entrada o salida de las Redes Neuronales, además de la información sobre parámetros como los pesos o los sesgos [12]. De esta forma, dicha estructura de datos abstractos permite realizar una manipulación y control muy sencillo de los datos almacenados. Añadido a esto, los tensores resultan ser tan importantes y relevantes debido a que son capaces de albergar datos de cualquier dimensionalidad como los narrays de Numpy, con la ventaja añadida de que estas estructuras de datos abstractos se pueden manejar en las GPUs, dando así un incremento en la velocidad de cálculo de operaciones de coma flotante (modelo cuda entre otros) [13].

Otra pieza importante en esta API serían los Módulos. Estos permiten utilizar esquemas ya definidos e incluso entrenados de forma rápida, además de construir módulos propios usando los ya definidos como piezas de tu modulo particular. Lo que hace que el desarrollador no solo no se tenga que preocupar de tener que desarrollar el algoritmo que describa el funcionamiento de dicho módulo, sino que también se simplifican otras fases como la retropropagación usado piezas como los optimizadores o los módulos autograd [14].

Por otra parte, otra característica importante más de esta librería es su uso de gráficos de cálculo dinámico, lo cual la diferencia de otras librerías como Tensor Flow que originalmente usaba grafos de cálculo estático. Los grafos de cálculo son las estructuras que representan el flujo de cálculos matemáticos en el proceso de aprendizaje de la Red Neuronal. Siendo así, encargados de realizar el cálculo de la propagación hacia delante y de la propagación hacia atrás. La diferencia entre dinámico y estático reside en el momento en el cual se va contrayendo o formando. Mientras que el dinámico se va construyendo en

tiempo de ejecución, el estático lo realiza en tiempo de compilación. Lo que deriva en que el cálculo dinámico aporta mayor flexibilidad [15].

Por último, con respecto a la librería Pytorch, caben mencionar características muy importantes como el uso de los Data Sets o los Data Loaders, los cuales aportan una gran sencillez en la carga de datos, pudiendo en pocas líneas tener disponible una conexión hacia un set de datos, con la posibilidad de manipularlo con un controlador que ayude a manejar el uso de dicho set para fases como el entrenamiento de la Red Neuronal. O el guardado de modelos ya entrenados, lo que aporta características de portabilidad, pudiendo trasladar el desarrollo completo de un entorno o proyecto a otro [16]. O, por último, la compatibilidad con otros lenguajes de programación como C++ o Java usando la herramienta TorchScript, que abre la puerta hacia la implementación en producción.

De esta forma, Pytorch con estas y más piezas se define como una herramienta, además de fácil de usar, versátil y flexible debido a su gran capacidad de poder crear desde los modelos más simples hasta modelos de alta complejidad.

La librería de Pytorch, además, se ayuda de otras bibliotecas adicionales de código abierto para extender sus utilidades y características. Algunas de estas librerías son TorchVision o TorchText. Ambas son librerías enfocadas en ciertos campos de aplicación, una por el lado del procesamiento de imágenes y visión artificial, y la otra por el procesamiento de lenguaje. Sin embargo, su relevancia en este trabajo es debido al posible uso de modelos, transformaciones u operaciones que se implementan en estas librerías y que han sido construidas a partir de las funcionalidades básicas de Pytorch [17].

Otra librería que se usa es Numpy, una librería de uso extendido y de código abierto que fue creada en 2005 por Travis Oliphant. Siendo una librería desarrollada para el alojamiento y la manipulación de datos de gran variedad de dimensiones. Sin embargo, dado al uso de tensores que, como se ha dicho previamente, siguen la línea de los narrays de Numpy, pero con la ventaja de poder ser operados por GPUs [18]. El uso de esta librería será ínfimo y será destinado a situaciones muy específicas o para acompañar el uso de la librería Matplotlib. Matplotlib es otra librería de código abierto, también, que será usada debido a que ofrece un kit de visualización de grafos con una gran compatibilidad con Numpy. En específico interesa el uso de Pyplot, un módulo de Matplotlib que aporta una interfaz parecida a la de MATLAB [19].

Finalmente, para la utilización de este conjunto de librerías es recomendable el uso de una máquina con disponibilidad de GPU. Por tanto, como todas las máquinas de uso personal no disponen de GPU. Se puede optar por opciones como GoogleColab o Kagle que proveen servicio de computación en la nube con máquinas que si disponen de GPU. Sin embargo, por la ley de protección de datos que se rige sobre los datos prestados por la empresa, todo modelo que use estos datos debe de ser desarrollado en local.

3 Diseño

En este apartado se explicará el diseño realizado de la parte práctica del presente TFG, donde se puede ver un desglose en dos secciones. Por un lado, está la sección de experimentación donde el autor a desarrollado algunos ejemplos usando los modelos de red neuronal FFNN, RNN y LSTM. Y por el otro lado, está la solución que intenta abordar este trabajo.

3.1 Experimentación con FFNN y RNN

Como se ha estado viendo en apartados anteriores de la sección *2. "Estado del Arte"*. Las FFNN y las RNN se destinan a ser usadas para el procesamiento de datos con características distintas. Mientras que los datos que se usan para alimentar a una red FFNN son datos aislados, los datos usados para alimentar a una red RNN son datos con dependencias entre ellos. De esta forma, cuando una FFNN procesa una entrada solo procesa los valores de dicha entrada, por ello su aislamiento. Por la otra parte, una RNN, no solo tendrá en cuenta los valores de la entrada, sino que también la posición temporal de ese dato de entrada, lo que quiere decir que también observará en qué momento se dio dicho dato y que otros datos le precedieron. Dando a ver que no solo inferirá patrones con respecto a los datos de entrada, sino que también con respecto a la dependencia de unos datos hacia otros.

Por tanto, esta sección se centra en hacer ver de forma práctica y con ejemplos las diferencias entre las FFNN y las RNN, dejando algo de espacio para las LSTM. A su vez, también se quiere presentar de una forma visual el cómo construir estos tipos de modelos de red neuronal con el uso de la libraría Pytorch.

Para lograr esto, se abordará el mismo problema de clasificación de imágenes para cada modelo de red neuronal (FFNN, RNN y LSTM). Clasificación en la cual se usa el DataSet público que proporciona Pytorch llamado FashionMNIST. Dicho DataSet contiene imagen de 28*28 pixeles los cuales representan un valor en la escala de grises. Dichas imágenes pueden pertenecer a una de las siguientes clases {0: "T-shirt/Top", 1: "Trouser", 2: "Pullover", 3: "Dress", 4: "Coat", 5: "Sandal", 6: "Shirt", 7: "Sneaker", 8: "Bag", 9: "Ankle boot"} las cuales se diferencian por su valor numérico. Asimismo, cada imagen está vinculada a una etiqueta que es su correspondiente valor numérico el cual indica a que clase pertenece la imagen. Finalmente, este DataSet cuanta con dos partes una de entrenamiento y otra de testeo. Por lo tanto, el modelo resultado deberá de poder clasificar de forma correcta que tipo de prenda o accesorio se muestra en cada imagen que se pase como entrada.

Añadido a esto, para probar las características y ver mejor las diferencias entre las redes FFNN y RNN (o LSTM), se termina esta sección de experimentación con la realización del mismo procedimiento para cada uno de los casos con la diferencia de la modificación de los datos que contiene el DataSet, como un caso alternativo. Esta modificación consiste en una permutación o un desordenamiento aleatorio de los diferentes pixeles que contiene cada imagen.

Teniendo como posible resultado la desaparición de cualquier posible orden natural y con ello cualquier relación de dependencia entre los datos del DataSet. Lo que en teoría significaría exagerar aún más el ejemplo de clasificación de imagenes con el uso del DataSet FashionMNIST para hacerlo más adecuado para un modelo FFNN que para un modelo RNN.

Por último, en el caso de los modelos de redes recurrente, se podría juntar este modelo con una secuencia de capas con diferentes funciones de activación, según como lo necesite el problema a resolver. Por tanto, se puede juntar la RNN con la FFNN anterior. Sin embargo, en este caso, se opta por una arquitectura con la presencia únicamente de la RNN dado a la naturaleza de este problema de clasificación y al carácter de ejemplo que tiene el caso.

3.1.1 Arquitectura Usada

Como se puede ver en este diagrama y como se ha especificado en los subapartados anteriores de este apartado. El diseño de estos ejemplos pasa por la obtención de los datos del DataSet FashionMNIST, la posible visualización de estos datos gracias a la librería Matplotlib y el uso de estos datos en un modelo de red neuronal para su entrenamiento y testeo usando para ello la librería Pytorch.

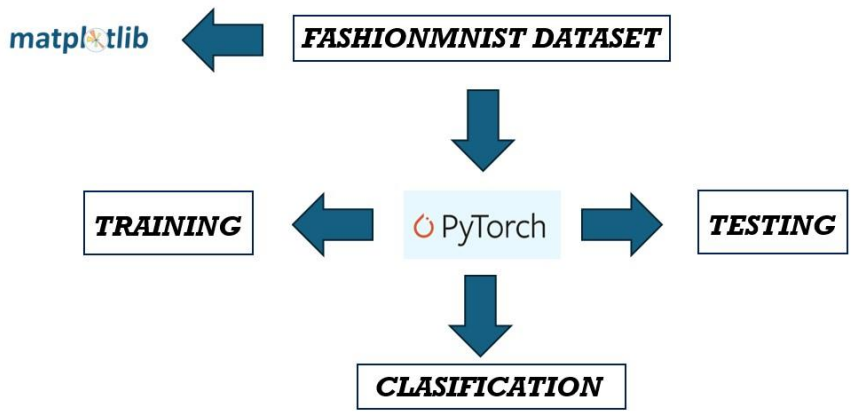


Figura 14. Gráfico de la arquitectura de los ejemplos de experimentación.

3.1.2 Metodología de Desarrollo

3.1.2.1 Redes o Modelos Usados

Para esta sección se han usado diferentes modelos dado que en esta parte se quiere mostrar cómo se trabaja con cada uno de los modelos y sus

peculiaridades entre otras cosas. Los modelos usados han sido el modelo FFNN, RNN y LSTM.

3.1.2.2 DataSets Usados

Para esta ejemplificación se ha hecho uso del DataSet público que proporciona Pytorch FashionMNIST.

3.1.2.3 Casos de Uso

Como se ha mencionado, esta implementación tiene como objetivo la experimentación y aprendizaje del alumno a la par que a la explicación práctica de las diferencias entre una red FFNN y una red recurrente. Por tanto, es una sección dedicada a la explicación de cómo se implementan y construyen las redes neuronales de cada clase.

3.2 Solución de Predicción de Ventas

Pasando al problema que se intenta resolver en este presente TFG. Para la realización de un modelo de red neuronal que prediga las ventas futuras de una empresa, se ha optado por usar un modelo basado en redes recurrentes LSTM. La razón por la que se ha escogido este modelo es por su gran rendimiento al usar secuencias temporales largas, además de su poca susceptibilidad hacía problemas de gradiente explosivo o desvanecedor. Asimismo, en este problema a diferencia de los ejemplos de experimentación se ha realizado una gran labor en el procesamiento, limpieza y adecuación de los datos para que estos pueden ser usados por el modelo.

Por otra parte, se ha realizado un proceso de entrenamiento y actualización, con testeo en cada una de estas partes. De esta forma se realiza un entrenamiento inicial del primer año y medio registrado dejando una pequeña parte de este conjunto de datos para realizar el testeo de dicho entrenamiento. Más adelante, para poder introducir esta sección que se usó como testeo, se introducen esos datos y más en la parte de actualización. Actualización la cual cuenta con una parte del set de datos para testeo también. Dicha actualización se haría de forma repetitiva dado que un modelo de red neuronal necesita de nuevos datos para poder inferir patrones nuevos y poder hacer predicciones de nuevos acontecimientos. Esto, deja como resultado un proceso en el cual, por ejemplo, con una frecuencia diaria se actualiza el modelo con nuevos datos del día para predecir así los del día siguiente. Luego, esas predicciones se comparan con el resultado real una vez pasado el día y se vuelve a actualizar el modelo. En el caso del problema que se plantea en este TFG debido a su naturaleza se hace de forma mensual y por ser un caso de ilustración se realiza una sola actualización en vez de una actualización repetida hasta llegar a los últimos datos, dado que solo se quiere hacer ver cómo sería el proceso.

Ahora, con respecto a los datos, se ha seguido una dinámica de minería y trabajo con el dato llamada CRISP-DM [50], que son las siglas de Cross-Industry Standard Process for Data Mining, en inglés. Esta metodología cuenta con seis fases involucradas en el entendimiento y preparación de los datos, además del desarrollo de modelos que usarán esos datos para poder analizar y prever futuros acontecimientos. Dichas fases con las que cuenta esta metodología son:

- **Comprensión del Negocio (Business Understanding):** Se define el contexto del problema y cuáles son los objetivos o requerimientos a conseguir.

- **Comprensión de los Datos (Data Understanding):** Una vez establecido el objetivo, es crucial analizar los datos que puedan ser necesarios para el desarrollo de la solución. Por tanto, en esta fase es requerida la realización de tareas como la identificación de datos, análisis de estos o verificación de su calidad.

- **Preparación de los Datos (Data Preparation):** Una vez recopilados los datos relevantes para el problema, se necesita realizar labores de ciencia de datos para la preparación y adecuación de estos. Lo que da la oportunidad de poder explotar y usar estos datos en las siguientes fases.

- **Modelado (Modeling):** Acto seguido se pasa al diseño y desarrollo del modelo que usara los datos previamente preparados. En el caso del problema que se intenta resolver en este TFG, se tendría el diseño y desarrollo de la red neuronal recurrente LSTM.

- **Evaluación (Evaluation):** Una vez terminado el modelo se usan los datos previamente preparados para probar y evaluar si el modelo cumple con los objetivos marcados en la fase inicial, además de si resuelve el problema planteado. Para ello se realizan fases de testeo en la que se evalúa la precisión entre otros parámetros.

- **Despliegue (Deployment):** Una vez evaluado, si el proyecto cumple los objetivos ya solo faltaría su puesta en producción para que pueda ser accedido por el usuario final al que está destinado.

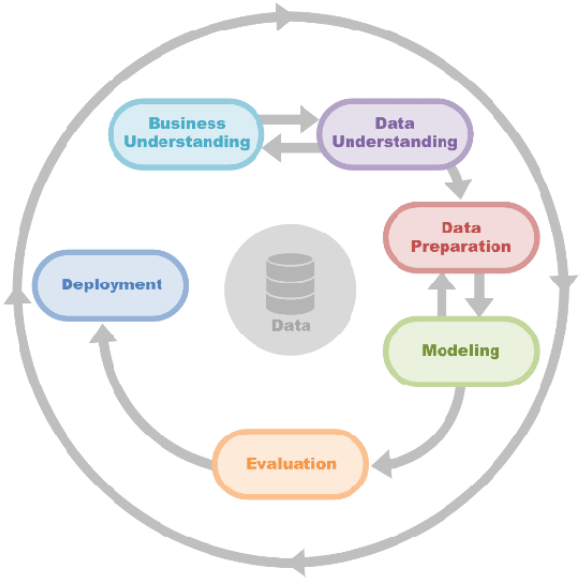


Figura 15. Gráfica explicativa del proceso CRISP-DM con sus seis fases [51].

En este presente TFG, durante el desarrollo e implementación de la solución para el problema de predicción de ventas se han realizado los cinco primeros pasos dejando fuera el proceso de despliegue.

3.2.1 Arquitectura Usada

El diseño de la solución pasa por el diagrama mostrado en la figura 16, el cual se sostiene con el uso de Python como lenguaje de programación. Primero se obtienen los datos de un fichero .csv para acto seguido ser procesados y adaptados, usando para ello librerías como pandas y numpy. De esta forma, se tiene como resultado la salida de un fichero .csv con los datos ya adaptados y la posibilidad de ir al siguiente paso. En este segundo paso se usa la librería Pytorch para todo lo que está relacionado con la construcción y trabajo con el modelo de red neuronal. Implicando con ello el entrenamiento y el testeo del modelo. Así mismo, también se obtienen graficas las cuales plasman el desempeño de la red neuronal en su entrenamiento. Finalmente, se obtiene también como resultado las predicciones de las ventas futuras.

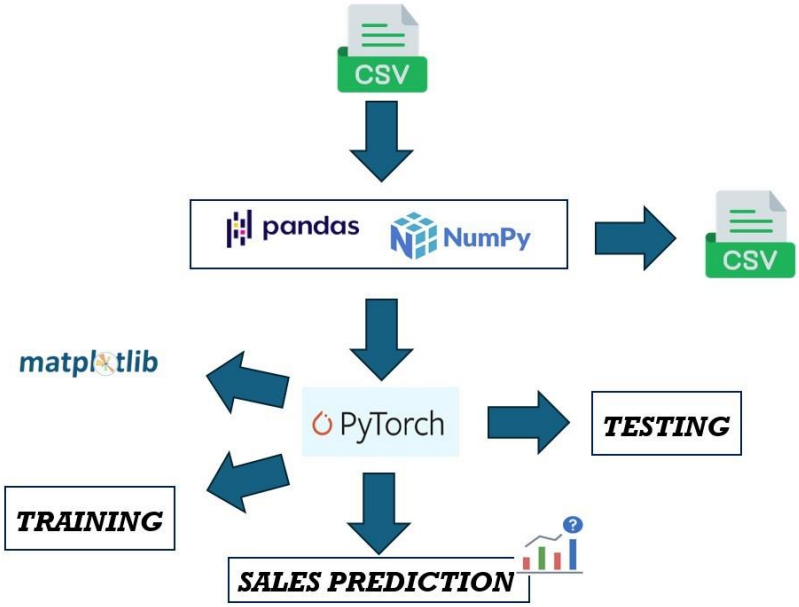


Figura 16. Gráfico de la arquitectura de la solución a la predicción de ventas.

3.2.2 Metodología de Desarrollo

3.2.2.1 Redes o Modelos Usados

Para esta parte se ha usado un modelo de red neuronal LSTM con las siguientes características:

Característica	Valor
Número de parámetros de entrada de cada timestep.	4
Número de variables de salida.	1
Número de capas ocultas.	2
Número de timesteps en la secuencia	257
Número de nodos en cada capa oculta.	256
Learning Rate.	0.00005
Longitud del lote de muestreo (batch).	32
Número de épocas para el entrenamiento.	50

Tabla 2. Especificación de hiperparámetros del modelo LSTM de la solución.

Se ha optado por un modelo LSTM debido al gran número de timesteps que tiene la secuencia temporal. Así mismo, la especificación de estos valores o hiperparámetros es crítica para poder tener un buen rendimiento del modelo. Por ello, se han asignado estos valores teniendo en cuenta recomendaciones y buenas prácticas. No obstante, para realizar la solución se ha elegido trabajar

sobre las categorías de productos en vez de los productos individualmente. Esto es debido a que el trabajar con los productos por separado significa manejar secuencias de 9575 timesteps de longitud lo que requiere muchos recursos. Por lo que, debido a las limitaciones del equipo del alumno, autor de este presente TFG, se ha optado por trabajar con las categorías reduciendo así la longitud de secuencia a 257 timesteps.

3.2.2.2 DataSets Usados

En este caso no se ha usado ningún DataSet ya conformado y público. Para esta sección se han tenido que adaptar los datos proporcionados por la empresa para acto seguido crear un DataSet propio con el que poder trabajar y usar con el modelo. Así mismo, como se ha dicho, para realizar la solución se trabaja con la categoría de los productos. Pero, además, la cantidad de variables/columnas usadas también ha sido delimitada por el mismo problema de limitaciones del equipo del alumno. Dado que ha más columnas/variables se manejen más pesado será el DataFrames o el grafo computacional a manejar en las operaciones, lo que significa que más consumo de recursos como memoria RAM se realizará. Por tanto, se ha optado por priorizar las variables más significativas con las que se pueda plantear una solución al problema.

3.2.2.3 Casos de Uso

Una vez realizada la implantación y el desarrollo de este modelo, se tiene como resultado una herramienta capaz de predecir el número de ventas de los diferentes productos de la empresa. Lo que supone una herramienta de gran poder para las empresas debido a que con su capacidad de predicción de ventas el empresario o gestor es capaz de controlar su stock e inversiones de forma más eficaz, además de otras cuantas ventajas.

4 Desarrollo

En esta sección se presentan las diferentes partes de código que componen el desarrollo de cada uno de los diseños que se explican en el apartado anterior "*3. Diseño*".

4.1 Experimentación con FFNN y RNN

El alumno, para poder obtener una cierta práctica previa a la realización de la solución acerca de la predicción de ventas que aborda este TFG, ha realizado ciertos ejemplos con los cuales ha podido experimentar, probar y observar el comportamiento de los modelos FFNN, RNN y LSTM. Ejemplos que el alumno plasma en este presente TFG no solo para enseñar las diferencias de comportamiento y características de estos modelos sino también para mostrar de forma detallada como construir y usar cada modelo. Por último, todo el código mostrado en este apartado sigue las pautas de diseño explicadas en el apartado anterior "*3.2 Experimentación con FFNN y RNN*".

4.1.1 Modelo FFNN

Para el problema de clasificación usando el modelo FFNN. Se empieza por intentar activar la GPU (Graphics Processing Unit) en el caso de que estuviera disponible. Lo que permitiría mover el modelo que se va a definir a la GPU para que esta unidad pueda operar con él. Y de esta forma, poder realizar los cálculos de este modelo de una forma más rápida.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"The device is: {device}")
```

Figura 17. Código de activación de la GPU.

Después, se sigue con el diseño del modelo FFNN y la especificación de las características que tendrá. Para ello, se hace uso de una clase propia la cual hereda de la superclase Module de Pytorch, heredando con sigo la estructura, los parámetros y las funciones de dicha clase Module. Con ello, en dicho modelo se usa la función *Flatten()* la cual permite convertir un array de N dimensiones en un array de una única dimensión. En este caso se usa para convertir la imagen representada por una matriz de 28 filas y 28 columnas en un vector de una única dimensión. Acto seguido, se usa la función *Sequential()* que permite indicar que tipos de capas se quieren encadenar. En esta secuencia y para este caso, aplicamos dos capas, una con 128 nodos y usando la función de activación Sigmoidea y otra con 64 y usando la función de activación ReLu. Entre estas capas, se usan capas de función de activación lineal dado que son las que permiten realizar cambios en cuanto a la magnitud del número de valores que se manejan, por ejemplo, la última realiza el comprimido de 64 valores a 10.

```
class MyModule(nn.Module):
  def __init__(self):
    super().__init__()
    self.flatten = nn.Flatten()
    self.sequence = nn.Sequential(
        nn.Linear(28*28, 128),
        nn.Sigmoid(),
        nn.Linear(128, 64),
        nn.ReLU(),
        nn.Linear(64, 10),
    )

  def forward(self, x):
    x = self.flatten(x)
    return self.sequence(x)

model = MyModule()
model = model.to(device) if device == "cuda" else model
```

Figura 18. Código de definición e inicialización del modelo FFNN.

Una vez definido el modelo FFNN, se cargan los datos sobre los DataSet FashionMNITS, uno para entrenamiento y otro para testeo.

```
train_data = datasets.FashionMNIST(
    root='./data',
    train=True,
    download= not os.path.exists("./data"),
    transform=ToTensor())

test_data = datasets.FashionMNIST(root='./data', train=False,
          download=os.path.exists("./data"), transform=ToTensor())
```

Figura 19. Código de carga de los datos de FashionMNIST.

Para ver de qué datos se compone el DataSet se puede ejecutar este fragmento de código el cual enseña nueve imágenes al azar de todas la que tiene el DataSet.

```
sample = plt.figure(figsize=(4,4))
column, row = 3, 3
for element in range(1, row * column + 1):
  img, label = train_data[random.randint(0, len(train_data))]
  sample.add_subplot(row,column, element)
  plt.axis("off")
  plt.imshow(img.squeeze(), cmap="gray")
plt.show
```

Figura 20. Código de visualización de los datos de FashionMNIST.

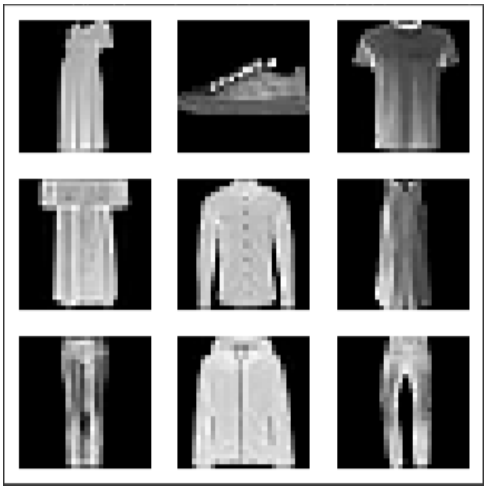


Figura 21. Muestra de imágenes del DataSet FashionMNIST.

Acto seguido, para la manipulación de estos datos, se hace uso de la clase DataLoader. Clase que permite obtener los datos por batches y realizar el fordward y backward propadation por lotes en vez de dato a dato, lo que supone ciertos beneficios. Se crea un DataLoader para test y otro para entrenamiento, como con los DataSet dado que ambas clases están vinculadas.

```
learning_rate = 0.01
batch_size = 64
epochs = 50

train_dataloader = DataLoader(train_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)
```

Figura 22. Código de creación de los DataLoaders.

Inmediatamente después, se implementa el proceso de entrenamiento del modelo. Para ello, primero se especifica el uso de la entropía cruzada como la función de error. Esta elección se debe a que su rendimiento y funcionamiento es adecuado para problemas de clasificación como el usado para esta sección [53]. Además, se decide usar el optimizador *torch.optim.SDG* dado que es uno de los indicados a usar cuando se trabaja con modelos FFNN. Llegado a este punto se realiza el proceso iterativo de obtención de lotes de muestras de entrada, predicción de la salida con esas muestras, cálculo del error de esas predicciones con respecto a la salida real, realización del backpropagation y actualización de pesos y bias con el resultado del backporopagation y el uso del optimizador. Cuando se terminan los datos de entrada y ya no se pueden obtener más lotes, significa que se ha completado una época. El proceso de entrenamiento termina cuando se termine de completar el número de épocas indicado.

Se recomienda usar diferentes marcas de progreso para poder observar cómo está funcionando el modelo. De esta forma se puede observar en qué medida el modelo está progresando y si lo está haciendo a mejor, ósea aprendiendo o a peor, desaprendiendo.

```
model.train()

error_function = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

error_per_epoch = []

for e in range(1, epochs+1):
  for num_batch, (X , Y) in enumerate(train_dataloader):
    if device == "cuda":
      X, Y = X.to(device), Y.to(device)

    #Fordward propagation.
    prediction = model(X)
    loss = error_function(prediction, Y)

    #Backward propagation.
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    if num_batch % 100 == 0:
                print(f"loss: {loss.item():>7f}  number of batch = {num_batch}")

  error_per_epoch.append(loss.item())
  print(f"[Epoch:{e}]: loss:{loss.item()}")
```

Figura 23. Código de entrenamiento del modelo FFNN.

Acto seguido, gracias a este código se puede obtener una interpretación gráfica del progreso del error durante el entrenamiento por cada época.

```
plt.plot(range(1, epochs+1), error_per_epoch, label="performance of error per epoch")
plt.xlabel("epoch")
plt.ylabel("error")
plt.title("performance of error per epoch")
plt.grid()
```

Figura 24. Código de visualización de la gráfica sobre el progreso del error del modelo.

Finalmente, se terminaría con la fase de testeo. Esta fase es parecida a la de entrenamiento realizando el mismo proceso iterativo. Sin embargo, aquí solo se itera una vez sobre el DataSet por lo que no se usan épocas y además no se realiza el backpropagation si no que solo el fordward propagation. Todas estas diferencias se deben a que no se quiere actualizar los parámetros del modelo si no medir la precisión de este.

```
model.eval()

error_amount = []

for num_batch, (X , Y) in enumerate(test_dataloader):
    if device == "cuda":
        X, Y = X.to(device), Y.to(device)

    #Fordward propagation.
    prediction = model(X)
    error_amount.append(error_function(prediction, Y).item())

print(f"The acurracy of the model is: {float(100 - np.mean(error_amount))}%")
```

Figura 25. Código de testeo del modelo FFNN.

4.1.2 Modelo RNN

Ahora se pasa a la clasificación con el uso de una RNN. En este caso se ha usado el mismo DataSet aunque de una forma distinta. Para poder tener secuencias de tiempo, en vez de tratar las imágenes como un vector unidimensional de 784 pixeles, se han usado las imágenes en su formato original como una matriz de 28 timesteps con 28 variables de entrada. No obstante, el código sigue la misma estructura y dinámica que en el caso anterior del uso de una FFNN. Sin embargo, hay ciertos cambios en algunos fragmentos que ahora se señalarán.

Para empezar, el modelo definido cambia completamente. Primero no se hace uso de la función *Flatten()*, lo que da a ver ese uso de la imagen en sus dimensiones originales de matriz 28*28 en vez de un vector de 784 valores como cuando se usaba el modelo FFNN. Segundo, se usa un modelo ya definido por Pytorch de RNN para después terminar usando una capa de nodos con función de activación lineal para poder amoldar el modelo a la forma que tiene la salida. En cuanto al modelo de RNN usado, se le tienen que especificar valores como la cantidad de variables que contiene un tiemsteps, la cantidad de nodos que contiene cada capa oculta, el número de capas ocultas, número de variables consideradas en la salida o el número de timesteps que contiene una secuencia.

Por otro lado, en lo que respecta a la propagación hacia adelante (fordward propagation) del modelo, se especifica el enlazado del modelo RNN con la última capa. Siendo el resultado del último timestep el seleccionado para ser la entrada de la capa lineal puesto que ese último timestep es el que tiene la información final del último hidden state.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"The device is: {device}")

input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
sequence_length = 28


class MyRNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers, num_classes):
    super().__init__()
    self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
    self.final_layer = nn.Linear(hidden_size, num_classes)


  def forward(self, x):
    out, _ = self.rnn(x)
    return self.final_layer(out[:, -1, :])

model = MyRNN(input_size, hidden_size, num_layers, num_classes)
model = model.to(device) if device == "cuda" else model
```

Figura 26. Código de definición del modelo RNN.

Definido ya el modelo, no se ve cambio con respecto al código del modelo FFNN hasta llegar a la sección de entrenamiento del modelo y la de especificación de hiperparámetros. Con respecto a estos hiperpárametros, se puede apreciar un gran cambio en el learning rate, se usa una tasa de aprendizaje bastante más baja que la usada en este mismo ejemplo, pero con una FFNN.

```
learning_rate = 0.00005
batch_size = 64
epochs = 50

train_dataloader = DataLoader(train_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)
```

Figura 27. Código de especificación de hiperpárametros del modelo RNN.

Y con respecto al proceso de entrenamiento, se puede observar que se sigue haciendo uso de la función de error CrossEntropy debido a que se usa como ejemplo un problema de clasificación, en este caso de imágenes. Pero, no obstante, se usa otro optimizador como es Adam debido a su mejor desempeño con modelos RNN o LSTM en comparación con otros optimizadores [54]. Como último cambio, entre la obtención del lote de datos y el comienzo del entrenamiento se realiza un necesario reajuste de la dimensión de estos lotes.

```
model.train()

error_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), learning_rate)

error_per_epoch = []

for e in range(1, epochs+1):
  for num_batch, (X , Y) in enumerate(train_dataloader):
    if device == "cuda":
      X, Y = X.to(device), Y.to(device)

    #Redimension de la entrada para ser coherente con la dimension pedida
    X = X.reshape(-1, sequence_length, input_size)

    #Fordward propagation.
    prediction = model(X)
    loss = error_function(prediction, Y)

    #Backward propagation.
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    if num_batch % 100 == 0:
            print(f"loss: {loss.item():>7f}  number of batch = {num_batch}")

  error_per_epoch.append(loss.item())
  print(f"[Epoch:{e}]: loss:{loss.item()}")
```

Figura 28. Código de entrenamiento del modelo RNN.

Por último, en cuanto al código del modelo RNN, en la parte de testeo del modelo se ven los mismos cambios que se han realizado en la parte de entrenamiento. El reajuste de la dimensión de los lotes.

```
model.eval()
error_amount = []
for num_batch, (X , Y) in enumerate(test_dataloader):
    if device == "cuda":
      X, Y = X.to(device), Y.to(device)

    X = X.reshape(-1, sequence_length, input_size)

    #Fordward propagation.
    prediction = model(X)
    error_amount.append(error_function(prediction, Y).item())

print(f"The acurracy of the model is: {float(100 - np.mean(error_amount))}%")
```

Figura 29. Código de testeo del modelo RNN.

4.1.3 Modelo LSTM

Con respecto al desarrollo del código que define al modelo LSTM la dinámica y estructura es exactamente igual a la del modelo RNN. Solamente se cambia una línea que es la que indica el modelo de Pytorch usado dentro de la clase que implementa el modelo personalizado.

```
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
sequence_length = 28

class MyLSTM(nn.Module):
  def __init__(self, input_size, hidden_size, num_layers, num_classes):
    super().__init__()
    self.rnn = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
    self.final_layer = nn.Linear(hidden_size, num_classes)


  def forward(self, x):
    out, _ = self.rnn(x)
    return self.final_layer(out[:, -1, :])

model = MyLSTM(input_size, hidden_size, num_layers, num_classes)
model = model.to(device) if device == "cuda" else model
```

Figura 30. Código de entrenamiento del modelo LSTM.

4.1.4 Caso Alternativo

Como caso alternativo se presenta el comportamiento de los mismos modelos, pero usando unos datos con las imágenes desordenadas. Para poder realizar este cambio en los datos, se realizan permutaciones aleatorias sobre los valores de las matrices que definen las imágenes. Permutaciones, las cuales se aplican a los datos del DataSet FashionMNIST descargados mediante la especificación de esta función de transformación como parámetro nombrado "*transform*".

```
# Generar un orden aleatorio de píxeles
random_order = np.random.permutation(28 * 28)

# Transformación para desordenar los píxeles de cada imagen
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Lambda(lambda x: x.view(-1)[random_order]),
    transforms.Lambda(lambda x: x.view(28, 28))
])

# Cargar el dataset MNIST con el desorden
train_data = datasets.FashionMNIST(root="./data", train=True
            , download= not os.path.exists("./data"), transform=transform)
test_data = datasets.FashionMNIST(root="./data", train=False,
            download= not os.path.exists("./data"), transform=transform)
```

Figura 31. Código de carga y transformación de los datos de FashionMNIST.

En el caso de la FFNN, en la definición del proceso de transformación de los datos no es necesario la realización de un redimensionado. Este redimensionado es realizado debido a que, al aplicar el desorden de los pixeles, esto deja como resultado un array de pixeles. Por ello, es necesario esta redimensión cuando se usa esta función de desorden en una RNN o LSTM, pero no cuando se usa en una FFNN. Desorden que como se puede observar en la figura 32, elimina cualquier orden natural de los timesteps y pixeles que componen la imagen.

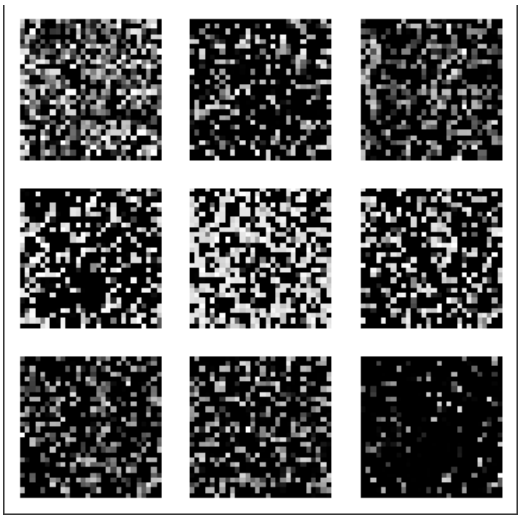


Figura 32. Muestra de imágenes del DataSet FashionMNIST alteradas.

4.2 Solución a la Predicción de Ventas.

Dejando los ejemplos atrás, en esta sección se verá un caso real con datos reales de las ventas de una empresa. En concreto, en esta sección se expone el proceso de desarrollo del diseño explicado en la sección anterior (3.2 Solución de Predicción de Ventas).

4.2.1 Adecuación de los Datos

Una parte muy importante cuando se trabaja con redes neuronales es la limpieza y preparación de los datos. Este proceso, si se realiza de forma correcta, hace que el proceso de definición de redes neuronales y su rendimiento mejore significativamente. En este caso se ha separado la adecuación de datos tipo fecha con respecto al resto de tipos de datos.

En cuanto a la parte de la limpieza de datos de tipo fecha, simplemente se ha buscado la especificación de que columnas son de tipo fecha, dado que pandas no es capaz de inferirlo al leer un documento. Así mismo, también se realiza la eliminación de toda entrada que no sigue un formato de tipo fecha correcto. Además de la transformación del formato a uno manejable, junto con la división de los valores de las fechas para poder ser procesados por el modelo de red neuronal. A todo este proceso también se añade la limitación por fecha de los datos a procesar.

```python
def dateTransformation(data_frame: DataFrame, date_column:str|list, initial_date:str,
                       final_date:str, format='%m/%d/%Y') -> DataFrame:

    data_frame[date_column] = pd.to_datetime(data_frame[date_column], errors='coerce')
    data_frame.dropna(subset=[date_column], inplace=True)

    data_frame[date_column] = data_frame[date_column].dt.strftime(format)
    data_frame[date_column] = pd.to_datetime(data_frame[date_column], format=format)

    initial_date = pd.to_datetime(initial_date, format=format)
    final_date = pd.to_datetime(final_date, format=format)
    data_frame = data_frame[(data_frame[date_column] >= initial_date)
                            & (data_frame[date_column] <= final_date)]

    data_frame = data_frame.copy()
    data_frame['Year'] = data_frame[date_column].dt.year
    data_frame['Month'] = data_frame[date_column].dt.month
    data_frame['Day'] = data_frame[date_column].dt.day
    data_frame.drop(date_column, axis='columns', inplace=True)

    return data_frame
```

Figura 33. Código de transformación de datos tipo fecha.

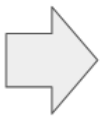
Por otra parte, con respecto a la transformación de los datos de los demás tipos, primero se empieza rellenando los valores NULL o NaN, minimizando así la perdida de datos. Luego se eliminan todas las columnas que no estén bien definidas y que pandas recoge con el nombre "Unnamed [0-9]+" cuando lee un fichero. Acto seguido se ve a que tipo de dato se puede cambiar cada columna si a Booleano, si no sea ha podido se intenta transformar a Float y si no a Integer. En el caso de no haberse podido cambiar el tipo de dato significa que la columna contiene valores String. Por tanto, debido a su carácter especial se toman tres estrategias según la información de clasificación de los datos que pueda aportar.

- **Eliminación de columna**: Si los valores de esta columna son muy dispersos y poco repetidos teniendo casi el mismo número de posibles valores que el número de entradas del conjunto de datos. Se opta por eliminar esta columna. Esto se debe a que la información que aporta esta columna es pobre dado que si cada entrada tiene un valor distinto esto no ofrece información alguna para poder encontrar patrones entre datos o patrones temporales de la secuencia. Pasa lo mismo si la columna solo tiene un posible valor.

- **Codificación Categórica**: En este caso el número de posibles valores no se acerca mucho al número de entradas del conjunto de datos. Por tanto, se opta por darle un valor numérico que clasifique a cada posible valor de la entrada [57].

- **Codificación One-Hot**: Cuando los datos que contiene una columna pueden optar a dos, tres o cuatro posibles valores, se elimina la columna y se añaden tantas columnas como posibles valores pueda tener la columna eliminada. De esta forma, cada columna lo que indica es si la entrada tiene ese valor o no (como un valor Booleano) [56].

id	color
1	red
2	green
3	blue
4	red



id	color	red	green
1	red	1	0
2	gree	0	1
3	blue	0	0
4	red	1	0

Figura 34. Proceso decodificación HOT-ONE.

```python
def dataTransformation(data_frame:DataFrame, one_hot_limit=5, categorical_limit=5) -> DataFrame:

    data_frame = data_frame.copy()
    for column in data_frame.columns:

        # fullfill all NaN values of the dataset
        data_frame.apply(lambda col: col.fillna(col.mean(), inplace=True) if col.dtypes != "object"
            and col.dtypes != "bool" else col.fillna(col.mode()[0], inplace=True))

        # converting booleans values in to int values
        if data_frame[column].dtype == "bool":                ##cambiar a opción modern
            data_frame[column] = data_frame[column].astype(int)

        # delete all the columns not well defined
        elif regex.match('Unnamed:\s\d+', column):
            data_frame.drop(column, axis='columns', inplace=True)
        else:
            try:

                data_frame[column] = pd.to_numeric(data_frame[column], errors='raise', downcast='float')

            except:
                try:

                    data_frame[column] = pd.to_numeric(data_frame[column], errors='raise', downcast='integer')

                except:

                    # classifying all data that couldn't be transformed
                    number_unique_values = data_frame[column].nunique()
                    if number_unique_values <= one_hot_limit and number_unique_values > 1:
                        data_frame = pd.get_dummies(data_frame, columns=[column], dtype=int)
                    elif number_unique_values > data_frame.shape[0] // categorical_limit or number_unique_values == 1 :
                        data_frame.drop(column, axis='columns', inplace=True)
                    else:
                        data_frame[column] = pd.Categorical(data_frame[column]).codes
    return data_frame
```

Figura 35. Código de transformación de datos.

Más adelante, debido a las características de los datos, se realiza una población de estos. Se busca que todo producto con un identificador aparezca por cada día indicando la cantidad vendida de ese producto en esa fecha en el caso de que se vendiese ese día. De esta forma es necesario agrupar los datos y ordenarlos para después añadir las filas con la información de los productos que no se hayan vendido en ciertas fechas. Teniendo así, como resultado una fila por cada producto por cada día, donde puede haber indicado una cantidad de productos vendidos o no haberlo y ser el valor 0.

```python
def dataRepoblation(dataframe:DataFrame, file_path:str, initial_date:str, last_date:str,
                    categories:str, format='%d/%m/%Y', delimiter=';', header=0, encoding='utf-8') -> DataFrame:

    initial_date = datetime.strptime(initial_date, format)
    last_date = datetime.strptime(last_date, format)

    first_months = pd.date_range(start=initial_date, end=last_date, freq='MS')
    last_months = pd.date_range(start=initial_date, end=last_date, freq='ME')

    if len(first_months) != len(last_months):
        raise RuntimeError("The dates expedited are not the first or last of month.")

    df_solution = pd.DataFrame()

    for i in range(len(first_months)):

        if os.path.exists(file_path):
            df_solution = pd.read_csv(file_path, delimiter=';', header=0, encoding='utf-8')

        d_range = pd.date_range(start=first_months[i], end=last_months[i])
        all_combinations = [
            {'id_category': np.float32(category), 'Day': np.float32(date.day),
             'Month': np.float32(date.month), 'Year': np.float32(date.year)}
            for category, date in product(categories, d_range)
        ]

        df_merged = pd.merge(pd.DataFrame(all_combinations), dataframe,
                             on=['id_category', 'Year', 'Month', 'Day'], how='left')

        df_solution = pd.concat([df_solution, df_merged], axis=0, ignore_index= True)
        df_solution['product_quantity'] =  df_solution['product_quantity'].fillna(0)

        df_solution = df_solution.sort_values(by=['Year', 'Month', 'Day', 'id_category'])
        df_solution.to_csv(file_path, sep=';', index=False)

    return df_solution
```

Figura 36. Código de repoblación de datos.

Llegado a este punto, dependiendo del problema, lo único que faltaría es la normalización de los datos. Esto significa cambiar los valores de los datos para que estén comprendidos bajo un rango como pueda ser el rango (0,1). Normalmente la normalización se aplica a los datos que servirán de entrada del modelo, pero también puede llegar a aplicarse para los de salida. Por ello, dado que valores muy lejanos a cero pueden sobrestimular algunas celdas o neuronas, es recomendable la realización de esta acción si la naturaleza del problema lo exige.

```python
def normalization(data_frame:DataFrame, min_value=0, max_value=1, outliner=False) -> DataFrame:
    #standar_scaler = StandardScaler()       # could produce noice so is not needed for RNN neural model type
    robust_scaler = RobustScaler()
    scaler = MinMaxScaler(feature_range =(min_value, max_value))
    numerical_columns = data_frame.select_dtypes(include=['int64', 'int32', 'int16', 'int8', 'float64', 'float32', 'float16']).columns

    if outliner:
        data_frame[numerical_columns] = pd.DataFrame(robust_scaler.fit_transform(data_frame[numerical_columns], columns=numerical_columns))

    data_frame[numerical_columns] = pd.DataFrame(scaler.fit_transform(data_frame[numerical_columns]), columns=numerical_columns)
    return data_frame
```

Figura 37. Código de normalización de los datos.

Finalmente, una vez se tienen los datos limpiados y adaptados se puede definir un DataSet el cual contenga esta información, lo que hace más manejable este conjunto de datos de cara a su posterior uso.

```python
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, dataframe, label_columns, sequence_length):
        self.input = torch.tensor(dataframe.drop(columns=label_columns).values, dtype=torch.float32)
        self.label = torch.tensor(dataframe[label_columns].values, dtype=torch.float32)

        self.input = self.input[:(self.input.size(0) // sequence_length * sequence_length)]
        self.input = self.input.reshape(-1, sequence_length, self.input.size(-1))


    def __len__(self):
        return len(self.input)

    def __getitem__(self, index):
        return self.input[index], self.label[index]

    def shape(self):
        return self.input.shape
```

Figura 38. Código de definición del DataSet a usar.

4.2.2 Construcción del Modelo LSTM

Una vez estén los datos limpios y adaptados para ser usados por el modelo de red neuronal, se pasa a la especificación de la arquitectura del modelo. Este modelo usa el módulo nn.LSTM() de Pytorch al cual se le pueden especificar valores como el número de parámetros que contiene la entrada o la salida, el número de nodos en cada capa oculta o el número de capas ocultas. Después se usa una capa lineal para poder condensar los datos y sacar el resultado o la salida. Adaptando la salida de la nn.LSTM() para que dé solo el último timestep y así poder realizar el enlace de la salida de la LSTM a la entrada de la capa lineal que conlleva.

Este no deja de ser un modelo el cual tiene como misión sacar la relación temporal entre las diferentes entradas y realizar predicciones como el número de ventas de una categoría de producto de días futuros. Sin embargo, en problemas más complejos como el análisis multivariable el modelo se puede complicar más al juntar una parte de recurrencia como una red LSTM con una segunda parte de capas densas FFNN. Mientras que la primera parte se encargaría de interpretar las relaciones temporales de la secuencia, la segunda se encargaría de realizar un análisis final de la salida de la parte anterior, dado que la salida de la primera parte no es una salida interpretable y final si no una especie de salida intermedia.

```python
class MyModule(torch.nn.Module):
  def __init__(self, input_size, hidden_size, num_layers, num_classes):
    super().__init__()
    self.rnn = torch.nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
    self.final_layer = torch.nn.Linear(hidden_size, num_classes)


  def forward(self, x):
    out, _ = self.rnn(x)
    return self.final_layer(out[:, -1, :])
```

Figura 39. Código de la definición del módulo predictor de ventas.

4.2.3 Proceso de Entrenamiento

Aquí se ve ya todo el proceso de entrenamiento al completo. Por tanto, se van a ir viendo los trozos de código que lo componen parte por parte. Mientras que todo el código mostrado en las dos secciones anteriores pertenecía al archivo nnlibrary_category.py, el código de esta sección pertenece al archivo trainig_category.py.

4.2.3.1 Adaptación de los Datos

Como podía ser de esperar la primera parte es la sección de adaptación y limpieza de datos, donde como se ve se llaman en secuencia a las diferentes funciones antes explicadas que se encargan de esta labor. Sin embargo, de entre todas estas funciones, no se llama a la función de normalización de los datos. Esto es debido a que ni los datos de entrada ni de salida requieren de esta normalización. Estos no presentan un rango de valores exageradamente amplio como para provocar problemas de gradiente explosivo o desvanecedor.

Acto seguido, para un mejor manejo de los datos, también se usan funciones de la librería panda las cuales ayudan a seleccionar que campos o variables se quieren usar o analizar, a agrupar los datos según ciertos campos y a ordenarlos. Y finalmente, se divide el DataFrame en un DataFrame destinado a entrenamiento y otro destinado a pruebas. Cabe mencionar que el proceso solo se hace una vez ya que se va guardando el resultado poco a poco en un fichero .csv.

```
if os.path.exists('./tmp/training_category.csv'):
    df = pd.read_csv('./tmp/training_category.csv', delimiter=';', header=0, encoding='utf-8')
    categories = pd.unique(df['id_category'])
else:
    df = pd.read_csv("../enterprise_data/ventas_date.csv", delimiter=';', header=0, encoding='utf-8')

    # saving some important information about the data
    categories = pd.unique(df['id_category'])
    categories = categories[~np.isnan(categories)]

    ## DATA CONVERSION AND FILTERING ##

    df = df[['id_category','product_quantity', 'date_add']]

    df = lib.dateTransformation(df, 'date_add', '01/01/2022','30/06/2023', format='%d/%m/%Y')

    df = df.groupby(['id_category', 'Year', 'Month', 'Day'])['product_quantity'].sum().reset_index()

    df = lib.dataTransformation(df)

    df = lib.dataRepoblation(df,'./tmp/training_category.csv', '01/01/2022', '30/06/2023', categories)

## SEPARATING THE DATA INTO TRAING AND TEST DATA ##

train_df = df[((df['Month'] != 6) | (df['Year'] != 2023))].copy()
test_df = df[(df['Month'] == 6) & (df['Year'] == 2023)].copy()
```

Figura 40. Código de transformación completa de los datos.

4.2.3.2 Especificación de Hiperparámetros para el Modelo

Se han seleccionado los valores de estos hiperparámetros de la red neuronal, que influyen fuertemente en su rendimiento, de forma consciente y en consonancia con los datos que se tienen. Algunos hiperparámetros como el número de capas de la red LSTM, el número de épocas o el número de nodos por cada capa oculta se han seleccionado de forma que sean lo suficiente para poder tener un rendimiento óptimo, pero no tan grandes como pare poder provocar un sobrecoste. Por otra parte, el hiperparámetro learning rate se ha ajustado de forma que no sea un valor grande que provoque una imposibilidad de convergencia de la red neuronal ni un valor tan pequeño que provoque que la red no pueda aprender. Finalmente, los hiperparámetros en relación con el número de variables de entrada y salida o la longitud de la secuencia temporal dependen directamente de cómo se presentan los datos y sus características. Mientras que el número de variables de entrada sería el número de columnas que tiene el DataFrame resultado de la limpieza, menos las columnas que expresan la salida de la red neuronal. El número de categorías de productos diferentes que se venden, en este caso, sería el valor de la longitud de la secuencia.

Por otra parte, cabe mencionar que también ha influido en la decisión de los valores de estos parámetros no solo la observación de cuáles son los valores recomendados como buenas prácticas sino también las capacidades y en este caso limitaciones de cómputo y memoria RAM del equipo del alumno. Por lo tanto, se han ajustado estos valores lo máximo que permitía el equipo para alcanzar los valores recomendados.

```
input_size = 4
num_layers = 2
num_classes = 1
sequence_length = len(categories)
hidden_size = 256
learning_rate = 0.00005
batch_size = 32
epochs = 50
```

Figura 41. Código de especificación de los hiperparámetros.

4.2.3.3 Creando el Modelo LSTM, los DataSet y el DataLoader

Para esta parte, primero se intenta activar la GPU si está disponible. Luego se crea el modelo LSTM llamando al módulo que se definió antes, especificando los diferentes hiperparámetros. Después de su creación, se intenta pasar este modelo a la GPU. Acto seguido se crea el correspondiente DataSet de cada uno, para luego terminar con la creación del DataLoader para cada DataSet.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"The device is: {device}")

model = lib.MyModule(input_size, hidden_size, num_layers, num_classes)
model = model.to(device) if device == "cuda" else model

train_dataset = lib.MyDataset(train_df, ['product_quantity'], sequence_length)
test_dataset = lib.MyDataset(test_df, ['product_quantity'], sequence_length)

train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size)
test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)
```

Figura 42. Código de creación del Modelo LSTM, los DataSet y el DataLoader

4.2.3.4 Bucle de Entrenamiento del Modelo

Para el entrenamiento, se sigue la secuencia marcada. Primero de todo se le indica al modelo que se prepare para el entrenamiento llamando a la función *model.train(),* lo que hace que el modelo tenga un mejor rendimiento en este proceso. Acto seguido se especifica la función de error. En este caso se usaría MSELoss (Mean Squared Error) en vez de la Entropía Cruzada puesto que mientras que el Error Cuadrático Medio funciona adecuadamente para regresiones la Entropía Cruzada funciona bien para clasificaciones [55]. Sin embargo, el alumno tras realizar varios intentos con esta función de error observó que el porcentaje de precisión era de 99.9999…%. Dicho porcentaje, muestra un valor algo alarmante puesto que en un modelo de redes neuronales raro es que la precisión sea un valor tan alto y se ajuste tan bien. La razón por la cual se da este valor es debido a que el modelo no se está ajustándose como tal si no que se está interpolando.

No obstante, la interpolación no se debe a un exceso de complejidad del modelo, sino que se debe a un desbalance de los datos usados. El problema de los datos que se usan está en que muchas categorías de productos no se venden en la gran parte de los días, lo cual provoca que muchas entradas del DatSet, más de un 98% de ellas, tengan como valor de cantidad de productos vendidos (ósea el valor real a predecir) el valor 0. Todo esto, termina provocando que el modelo al ver que hay una gran dominancia de un valor de salida, este no aprenda y simplemente saque ese valor como salida para toda entrada posible, lo cual asegura un alto porcentaje de acierto.

Como solución o forma de mitigación de este problema, se usa la función de error HuberLoss [58]. Función de error robusta y poco sensible a valores pequeños como los cercanos a 0 o valores muy grandes. Por tanto, esta función de error es una buena opción para intentar resolver este problema dado que lo que hará es darles más valor o relevancia a las entradas de datos con un valor de cantidad de productos vendidos diferente a 0. Lo que se traduce en que las entradas con valor distinto a 0 tienen un mayor peso y por tanto efecto en el cambio de los parámetros del modelo, cosa que las entradas con valor 0 como valor de salida no.

Después de especificar la función de error a usar, se especifica el optimizado a usar, en este caso Adam por su buen rendimiento con redes RNN o parecidas. Llegados a este punto, se realiza el bucle de entrenamiento donde se recorren todas las filas del DataSet de entrenamiento por cada época. Dentro de estas épocas, se obtienen de forma repetida lotes de muestras para realizar la predicción, la comparación de la predicción con el dato real, el cálculo del error y la retroprpagación.

Por último, se realizan dos funciones cruciales *optimizer.step()* y *optimizer.zero_grad()*. Mientras que el primero sirve para la actualización de parámetros como los pesos o los bias de cada nodo de la red neuronal. El segundo tiene la labor de limpiar la información acumulada del desempeño de los gradientes por cada lote. De esta forma el siguiente lote a procesar y usar no se ve afectado por la información acumulada de los gradientes del lote anterior. Aislando con ello, su procesamiento y su fase de entrenamiento.

```
model.train()
error_function = torch.nn.SmoothL1Loss()
optimizer = torch.optim.Adam(model.parameters(), learning_rate)
error_per_epoch = []

for e in range(1, epochs+1):
  for num_batch, (X , Y) in enumerate(train_dataloader):
    if device == "cuda":
      X, Y = X.to(device), Y.to(device)

    #Fordward propagation.
    prediction = model(X)
    loss = error_function(prediction, Y)

    #Backward propagation.
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    # a control mark every certain number of batch done.
    if num_batch % 10 == 0:
                print(f"loss: {loss.item():>7f}  number of batch = {num_batch}")

  error_per_epoch.append(loss.item())
  print(f"[Epoch:{e}]: loss:{loss.item()}")
```

Figura 43. Código de entrenamiento del modelo de predicción de ventas.

Así mismo, para tener una visualización más clara, se crea esta gráfica de Matplot para ver la información del progreso del error cometido por la red neuronal por cada época.

```
plt.plot(range(1, epochs+1), error_per_epoch, label="performance of error per epoch")
plt.xlabel("epoch")
plt.ylabel("error")
plt.title("performance of error per epoch")
plt.grid()
plt.show()
```

Figura 44. Código de visualización de gráfica sobre el error del modelo producido por época

4.2.3.5 Bucle de Testeo del Modelo

Para esta sección se realiza el mismo proceso que en el entrenamiento, con la diferencia de que no se realiza ni la iteración por épocas ni la retropropagación puesto que el objetivo de esta sección es ver la precisión de la red neuronal.

```
model.eval()
error_amount = []

with torch.no_grad():
  for num_batch, (X , Y) in enumerate(test_dataloader):
      if device == "cuda":
        X, Y = X.to(device), Y.to(device)

      #Fordward propagation.
      prediction = model(X)
      error_amount.append(error_function(prediction, Y).item())


print(f"The acurracy of the model is: {100 * float(1.0 - np.mean(error_amount))}%")
```

Figura 45. Código de testeo del modelo de predicción de ventas.

4.2.3.6 Guardado del Estado del Modelo

Cuando se implementa una red neuronal esta no se crea y entrena cada vez que se va a usar. Una vez que se ha entrenado se guarda su información de estado o la información de parámetros como los pesos y el bias de cada nodo. Esto te permite cargar el modelo de forma rápida y usarlo o incluso actualizar su estado entrenando la red con nuevos datos.

```
torch.save({
      'epochs': epochs,
      'model_state': model.state_dict(),
      'optimizer_state': optimizer.state_dict(),
      'loss': loss,
}, './modelstate/trained_category_model_state.pth')
```

Figura 46. Código de guardado del estado del modelo de predicción de ventas.

4.2.4 Proceso de Actualización

Como se menciona en una sección anterior, en el apartado *4.2.3.6 "Guardado del Estado del Modelo"*, al guardar el estado del modelo se puede después actualizarlo entrenándole con más datos. En este caso de predicción de ventas de una empresa, se puede haber estado durante un mes recopilando información que luego puede ser usada para entrenar y actualizar la red neuronal.

Este proceso de actualización es muy importante para las redes neuronales. Las redes neuronales tienen tanto conocimiento como lo que puedan interpretar de los datos que se usan para su entrenamiento. Por tanto, no es lo mismo usar un vasto set de datos que usar un fichero Excel con quinientas entradas. Cuantos más datos se usen y más variados o diferentes sean estos, más patrones puede hallar la red neuronal y por tanto mejor capacidad de predicción puede tener.

En este caso de predicción de ventas, al actualizar la red neuronal con los datos de un nuevo mes, el modelo puede que encuentre nuevos patrones de ventas debido a que en el mes hay vario días festivos, se hacen regalos o por otras posibles razones. Pero, si después de esta actualización se van realizando más actualizaciones progresivas puede que la red neuronal encuentre nuevos patrones que no podía inferir con los datos que disponía anteriormente.

Por tanto, para la actualización de la red neuronal se realizan los mismos pasos que con la creación y entrenamiento inicial de esta: limpieza de datos, preparación del modelo, creación del DataSet, creación del correspondiente DataLoader, entrenamiento del modelo y testeo del modelo. Sin embargo, se añade un paso que es la carga de la red neuronal y su estado que previamente fue guardo. Por otra parte, la especificación de los hiperparámetros puede que cambie o puede que no, eso depende de cómo se presenten los datos, pero al final sigue siendo misma filosofía.

```
error_function = torch.nn.SmoothL1Loss()
optimizer = torch.optim.Adam(model.parameters(), learning_rate)

checkpoint = torch.load('./modelstate/trained_category_model_state.pth')
model.load_state_dict(checkpoint['model_state'])
optimizer.load_state_dict(checkpoint['optimizer_state'])
loss = checkpoint['loss']
```

Figura 47. Código de carga del estado del modelo de predicción de ventas.

5 Resultados y evaluación

En esta sección se presentarán los resultados del desarrollo mostrado en el anterior apartado además de ser comentados y analizados.

5.1 Resultados y Evaluaciones de las Experimentaciones

5.1.1 Resultado del modelo FFNN

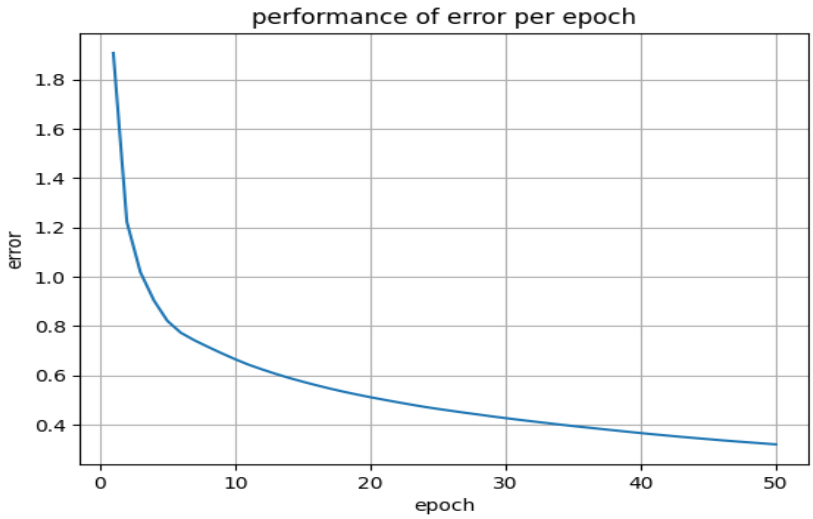


Figura 48. Gráfica del progreso del error por cada época del modelo FFNN.

The acurracy of the model is: 99.58300265555928%

5.1.2 Resultado del modelo RNN

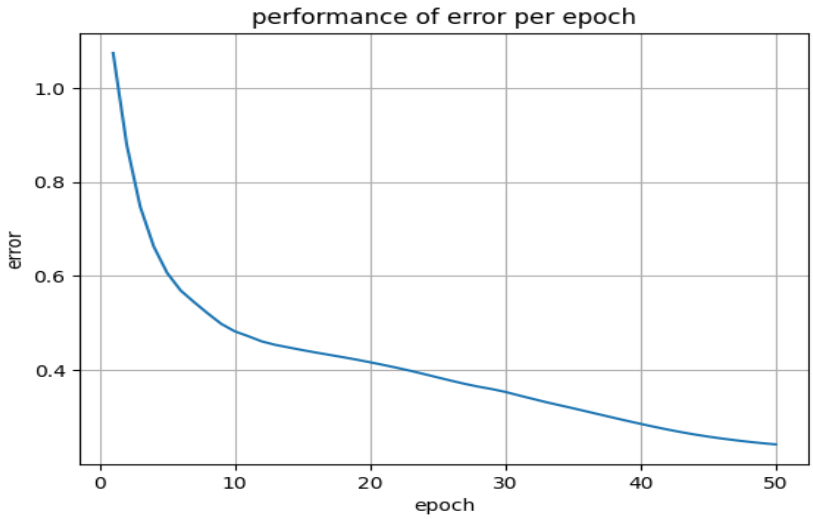


Figura 49. Gráfica del progreso del error por cada época del modelo RNN.

The acurracy of the model is: 99.55317185771693%

5.1.3 Resultados del Modelo LSTM

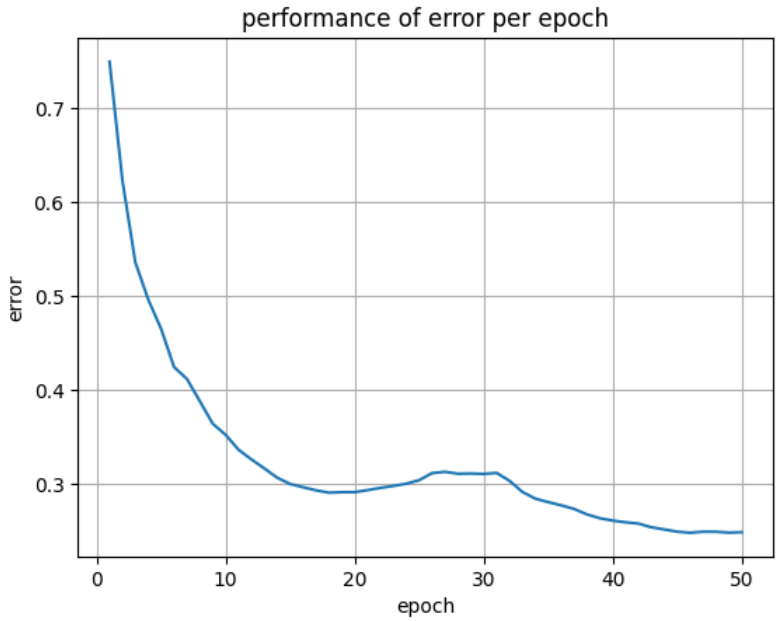


Figura 50. Gráfica del progreso del error por cada época del modelo LSTM.

The acurracy of the model is: 99.62968378480832%

5.1.4 Resultados del Modelo FFNN en el Caso Alternativo

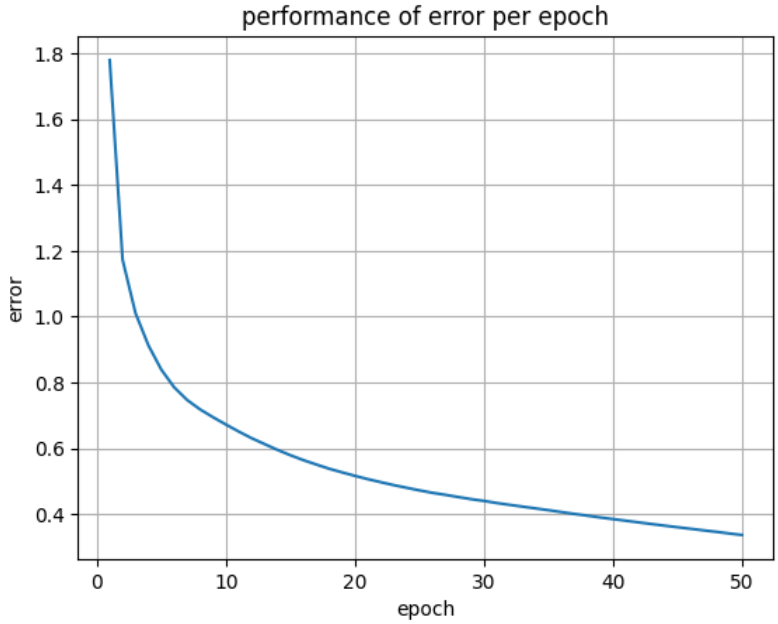


Figura 51. Gráfica del progreso del error por cada época del modelo FFNN en el caso alternativo.

The acurracy of the model is: 99.58056608241075%

5.1.5 Resultados del Modelo RNN en el Caso Alternativo

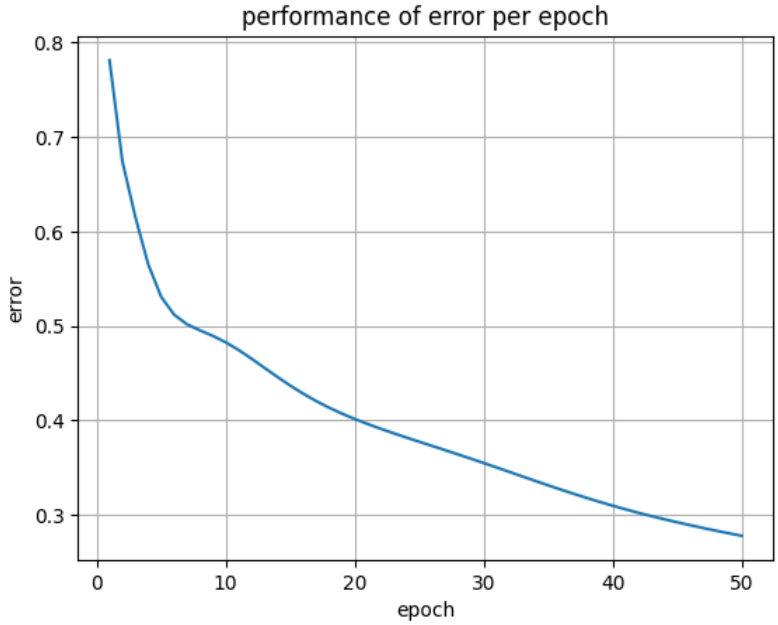


Figura 52. Gráfica del progreso del error por cada época del modelo RNN en el caso alternativo.

The acurracy of the model is: 99.59023499023762%

5.1.6 Evaluación de Resultados de la Experimentación

Con los resultados obtenidos se puede comprobar que no solo los resultados del modelo FFNN y RNN son idénticos, sino que esos resultados también son iguales cuando se aplica el desorden de los datos. Esto hace ver que los modelos de redes neuronales recurrentes (RNN o LSTM) son capaces de hacer las mismas labores que una red FFNN. Sin embargo, debido a que estos modelos recurrentes están diseñados para resolver problemas donde los datos tienen dependencia temporal entre ellos, como la traducción, estos terminan siendo más complejos que un modelo FFNN.

Por tanto, para una labor como es la clasificación de imágenes donde el modelo FFNN es capaz de resolver el problema de forma correcta. El usar modelos más complejos como una RNN hace que el gasto de recursos como computo o tiempo sea mayor sin necesidad de ello. En otras palabras, se están gastando recursos de forma innecesaria. Esto se puede comprobar con la diferencia de tiempo de entrenamiento cuando ambos modelos cuentan con condiciones muy parecidas. Mientras que el modelo FFNN tarda 8 minutos el modelo RNN tarda 28 en completar el proceso de entrenamiento. También hay una diferencia significativa en cuanto a uso de memoria RAM, mientras que el modelo FFNN hace uso de 1.5 GB de memoria RAM, el modelo RNN hace uso de casi 9 GB. En conclusión, se pueden usar modelos tan complejos como un modelo RNN o

incluso otros más complejos como pueda ser el LSTM. Sin embargo, siendo la IA y en concreto las redes neuronales una tecnología tan demándate de recursos, lo conveniente sería usar aquel modelo que pueda dar un rendimiento óptimo junto con unos resultados correctos. Por lo cual, en este ejemplo el uso del modelo FFNN debería de ser la opción correcta.

Otra cuestión a destacar es la diferencia en la asignación de un valor para el learning rate. Como se ha explicado en la sección *2. "Estado del Arte"*, los modelos RNN o LSTM son muy susceptibles a los cambios en sus parámetros como los pesos o bias y esto es debido a su carácter de recurrencia. El peso que se usa para la entrada de retroalimentación de una neurona o celda (en el caso de LSTM) con sigo misma es un valor que influye mucho en el desempeño de esta, debido a que su aparición en la fórmula que define a la red neuronal es muy repetida. Por ello, con la premisa de que ambos modelos convergen de forma correcta, mientras que el modelo RNN cuenta con un valor de learning rate igual a 0.00005, el modelo FFNN cuenta con un valor igual a 0.01. Valores los cuales, quizás, hagan ver que el modelo FFNN es más simple que el modelo RNN.

Por último, cabe mencionar que ambos modelos han conseguido una precisión aproximada de un 99.58%. Una precisión buena, pero quizás un poco demasiado alta lo que puede indicar que el modelo eta sobreajustado. Sin embargo, el alumno a realizado pruebas con diferentes combinaciones de hiperparámetros intentando evitar este problema. Por tanto, puede ser que no sea un problema de sobreajuste y simplemente que el modelo es muy preciso.

5.2 Resultados y Evaluación de la Solución de Predicción de Ventas

5.2.1 Resultado del Entrenamiento

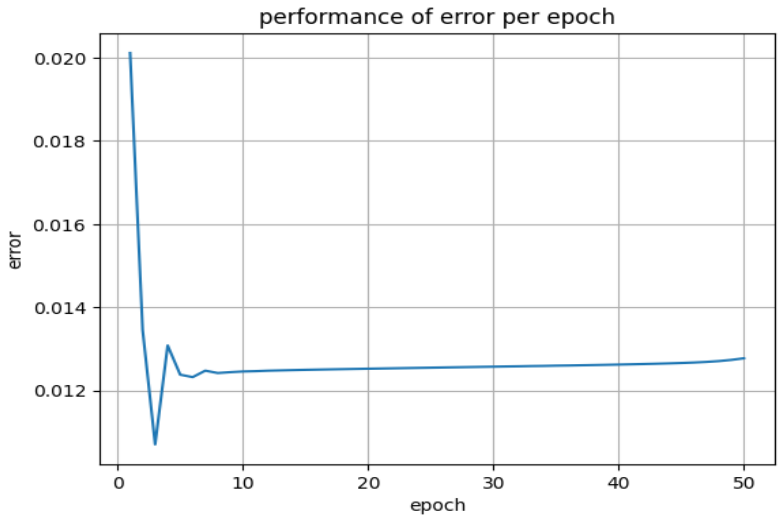


Figura 53. Gráfica del progreso del error por cada época del modelo de predicción de ventas en el entrenamiento.

The acurracy of the model is: 64.25440609455109%

The acurracy of the model is: -4.310693524100562%

5.2.2 Resultados de la Actualización

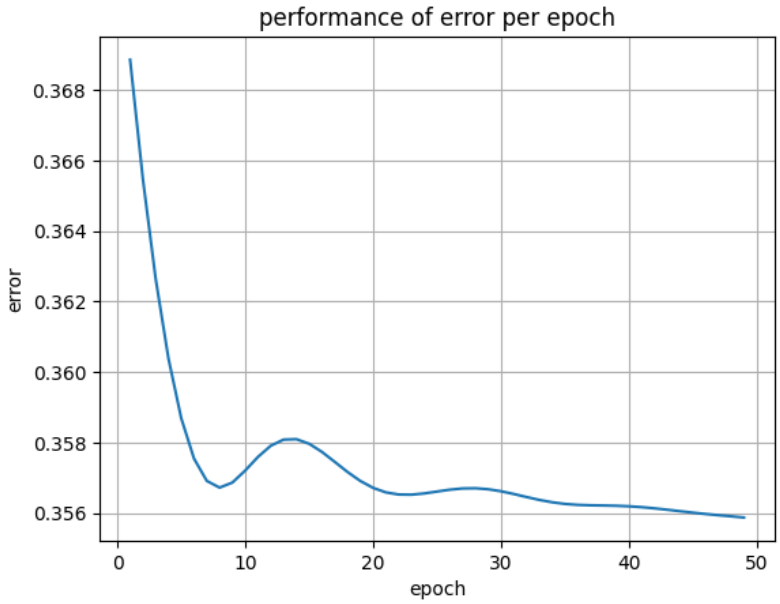


Figura 54. Gráfica del progreso del error por cada época del modelo de predicción de ventas en la actualización.

The acurracy of the model is: 75.13990253210068%
The acurracy of the model is: 0.9552154276106117%

5.2.3 Evaluación de Resultados de la Solución

Viendo los resultados obtenidos del modelo de predicción de ventas, se saca como conclusión que el desbalanceo de los datos afecta gravemente al rendimiento de dicho modelo. Como ya se ha comentado anteriormente. Frente a las munchas entradas de datos que indican la ausencia de ventas de una categoría de producto para una fecha concreta con un valor 0 para la variable de cantidad de productos vendidos. Hay muy pocas entradas que tengan un valor diferente a 0 y que indiquen que haya habido ventas para una cierta categoría de producto en una cierta fecha. Por lo cual, se está ante un caso en el que el valor 0 en la variable cantidad de productos vendidos domina con más de un 98% de aparición en los datos.

Este fenómeno provoca que el modelo no aprenda. El modelo intentará ajustar los parámetros como pesos y bias para que según que valores de entrada reciba pueda dar un resultado correcto por lo aprendido del set de datos de entrenamiento. Sin embargo, como en este caso, si casi siempre la salida es el valor 0 el modelo se ajustará de tal forma que siempre dará ese valor como

salida, lo que provoca que el modelo no aprenda de las entradas con valor de salida distinto a 0. Además, luego en la fase de testeo se tendrá un valor engañoso extremadamente alto (99.999…%) debido en gran parte a que cuanto mayor sea la dominancia del valor de salida mayor será la precisión. El modelo solo se equivocaría en aquellas ocasiones donde el valor de salida es distinto al valor dominante.

Para poder evitar estos valores de precisión engañosos y solucionar un poco el problema de desbalanceo de datos, se usa la función de error HuberLoss [58]. Una función más robusta que el Error Cuadrático Medio dado que ante valores muy pequeños, cercanos a cero, o valores muy grandes esta función reduce el cambio que puedan provocar en el cálculo de errores dejando más protagonismo al resto. De esta forma, en este caso, lo que se consigue es que se le dé más peso o importancia al error producido por entradas con valor de salida diferente a 0. Por ello, cuando se hace uso de la función de error HuberLoss los valores de precisión bajan hasta el 70-75%. Así mismo, se ha calculado la precisión del modelo teniendo en cuenta solo entradas que tengan un valor de salida distinto a 0. Lo que ha dado como resultado que con el uso de una función de error como el Error Cuadrático Medio la precisión sea un acumulado de errores cometidos llegando a valores como -227.6796%. No obstante, con el uso de la función de error HuberLoss este valor a podido no solo llegar a valores negativos cercanos a cero, sino que ha podido llegar a valores positivo muy bajos. Lo que da a ver que la función de error HuberLoss ha logrado que el modelo aprenda algo de esas entradas con valor de salida diferente a 0. Sin embargo, un valor de 0.9552154276106117% de precisión no es un valor aceptable para un modelo de red neuronal. En este caso, se podrían usar otras técnicas como la aplicación de pesos en los cálculos de errores, pero aun así podría no ser suficiente.

Finalmente, se tiene como hipótesis que el problema por el cual se está dando este resultado es debido a que se está intentando aplicar una tecnología que usa una vasta cantidad de datos cuando en realidad la empresa no tiene un volumen de ventas suficiente como para poder generar esa cantidad de datos que se necesita. Por tanto, una de las posibles vías a seguir es buscar otras opciones tecnológicas acordes a la situación y el problema.

6 Conclusiones

Para esta sección se van a explicar los resultados y las conclusiones obtenidas en este trabajo tanto de forma general como en base a cada objetivo que se planteó desde un inicio.

6.1 Resultados Globales

Tras la realización de este presente TFG el autor ha conseguido entender y definir los primeros pasos y las primeras ideas de las redes neuronales. Desde los comienzos de este campo hasta la invención de las redes recurrentes. De esta forma, en este trabajo se realiza una explicación detallada de la teoría que rodea a las redes neuronales, bajo el entendimiento del autor. Por tanto, el autor ha podido realizar explicaciones sobre conceptos del área de las redes neuronales como son: El Ajuste, La Interpolación, El Backpropagation o modelo de redes neuronales como las FFNN, las RNN o las LSTM.

Mas allá de lo teórico y haciendo hincapié en lo práctico, el autor ha logrado dos cosas. Primero, ha podido desarrollar un ejemplo en el que se explica como implementar un modelo FFNN y un modelo RNN con el uso de la librería Pytorch. Explicando en cada caso los detalles y peculiaridades en la implementación de cada modelo debido a la naturaleza y características de estos. Además, el autor ha usado esa misma sección para experimentar el rendimiento de cada uno de los modelos ante un mismo caso. Experimento con el que el autor ha llegado a la conclusión de que los modelos FFNN tienen una gran capacidad para inferir patrones entre los datos de entrada, aunque estos no sigan un orden natural. Y que los modelos RNN, aunque también pueden inferir patrones entre los datos de entrada su rendimiento en esta labor se ve mermado debido a su complejidad frente a los modelos FFNN. Complejidad, que necesita para realizar la labor por la cual fue inventado este modelo, la inferencia de patrones en las relaciones de dependencia temporales entre diferentes entradas de datos. Lo cual hace ver que, en el ejemplo, el uso de una RNN es incorrecto mientras que el de una FFNN es correcto para el caso.

Y segundo, el autor presenta el desarrollo de una red neuronal LSTM capaz de predecir el número de ventas de un producto en días futuros. De esta forma, presenta los diferentes pasos que se han realizado en la implantación de esta solución. Desde la preparación de los datos hasta la obtención de las predicciones y la realización del proceso de pruebas. Además de ello, el autor también presenta el desarrollo del proceso de actualización de la red neuronal. Permitiendo de esta forma disponer de una herramienta de evaluación predictiva de las ventas de la empresa que puede actualizarse evitando con sigo su desfase. No obstante, al haberse presentado problemas con los datos prestados por la empresa el autor ha podido investigar sobre este posible problema y ver que soluciones comunes se pueden realizar. Sin embargo, el alumno no ha conseguido una solución para unos datos pobres.

6.2 Resultados por Objetivos

Desde los inicios en el desarrollo de este trabajo se realizó una planificación la cual describía un orden cronológico de objetivos. Teniendo en cuenta esos objetivos que se especificaron en un comienzo se puede expresar lo siguiente:

- **Aprendizaje Teórico:** Uno de los primeros objetivos cumplidos fue la adquisición de información y el aprendizaje sobre el tema el cual desarrolla este trabajo (las redes neuronales). De esta forma, se obtuvo información de fuentes muy variadas como puedan ser articulo (papers), manuales, artículos web, fragmentos de libros... lo que ayudo a poder tener una visión desde los más profunda hasta lo más general. Tras la lectura de estas fuentes, el alumno pudo obtener un conocimiento detallado sobre las diferentes redes neuronales de las que se hablan en este trabajo FFNN, RNN y LSTM.

- **Práctica con herramientas:** Tras ese comienzo teórico el alumno paso a realizar diferentes pruebas con las tecnologías y herramientas usadas Pytorch, Pandas, Matplotlib, Numpy... (más información en el apartado *1.2 "Tecnologías Usadas"*). Para ello, el alumno hizo uso de manuales y cursos online para poder aprender como realizar todos los procesos necesarios para poder implementar una red neuronal.

- **Desarrollo del modelo de datos:** Una vez hecha la práctica el alumno comenzó con la realización e implementación de la limpieza y preparación de los datos. Lo que supuso analizar los datos para interpretar que información es la relevante y que debe de usarse, que formato deben de seguir cada uno de los datos o como se deben de cambiar para poder ser usados por la red neuronal. Un proceso el cual conlleva el mayor esfuerzos en este tipo de trabajos.

- **Desarrollo de la red neuronal:** Posteriormente el alumno desarrolló el modelo de red neuronal para la solución observando y analizando que parámetros o arquitectura era la más adecuada. Más adelante se desarrollaron el proceso de entrenamiento para el ajuste de la red neuronal y el de testeo para poder medir la precisión de ella.

6.3 Conclusiones del Estudiante

La realización de este presente TFG ha supuesto en el estudiante un primer contacto cercano en el desarrollo de redes neuronales. Habiendo sido de esta forma una buena oportunidad para adquirir y aprender diversos conceptos teóricos y práctico en una disciplina de emergente crecimiento que se posiciona actualmente como una de las ramas de la informática de mayor impacto social.

7 Análisis de Impacto

En este capítulo se realizará un análisis del impacto potencial de los resultados obtenidos durante la realización del TFG en diferentes contextos:

- Personal: En relación con lo ya comentado en el apartado 3.3 Conclusiones del Estudiante, de forma personal el estudiante y autor de este presente trabajo ha tenido la oportunidad de poder realizar un proyecto el cual abarca uno de los temas emergentes y con mayor impacto social actualmente. Pudiendo adquirir destrezas y conocimientos que más adelante prevé necesarios para cualquier profesional de la industria tecnológica.

- Empresarial: Este presente trabajo les proporciona a las empresas una base ideológica y una muestra práctica de una herramienta capaz de dar la información necesaria a las empresas para poder ajustar el nivel de producción correcto para abastecer la demanda de los clientes. De esta forma se plantea una herramienta con la que las empresas puedan evitar fenómenos conocidos como *stock muerto* o *exceso de inventario*. Esto se traduce en beneficios para las empresas al reducir los costes de producción y almacenamiento, al no tener que producir productos que estén destinados a quedarse en los almacenes. Por tanto, en lo relacionado al tema empresarial esta solución puede tener un cierto impacto en el objetivo 9 "Industria, innovación e infraestructuras" al ser una idea que promueve el apoyo a la innovación en el tejido empresarial.

- Económico: El impacto económico se limita a la reducción de costes al poder disminuir el *stock muerto* o *exceso de inventario*. Consiguiendo de esta forma que la energía, las materias primas y el tiempo invertido en la producción de estos productos se destinen a otros que si van a llegar al consumidor. De esta forma, se puede ver esta solución como una oportunidad de rentabilizar lo máximo posible cada unidad de energía tiempo y materia usadas.

- Medioambiental: Este ajuste de las empresas en sus niveles de producción también puede provocar una gran reducción en la generación de desechos, la contaminación y el uso de materias primas usadas en los procesos de fabricación, transporte y reciclaje de aquellos productos que no terminan de llegar a los consumidores. De esta forma, esta solución puede provocar un gran impacto en medidas de la agenda 2030 como son los objetivos 12 "Producción y consumo responsables", 13 "Acción por el clima", 14 "Vida submarina" y 15 "Vida de ecosistemas terrestres".

Ahora, esta solución como toda solución basada en inteligencia artificial demanda un gran uso de procesamiento, lo que provoca también un gasto considerable de energía. Por tanto, como con toda herramienta, su uso sin control y de forma desmedida puede provocar efectos perjudiciales.

8 Bibliografía

[1] J. Humberto y A. Peña, *Estado del Arte en Inteligencia Artificial y Ciencia de Datos*. MX: IPN, 2019.

[2] R. S. Rosenberg, *The Impact of Artificial Intelligence on Society*. Department of Computer Science, University of British Columbia, 1988.

[3] W. S. McCulloch y W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, pp. 115–133, 1943.

[4] P. J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. dissertation, Harvard University, 1975.

[5] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, Oct. 1990, doi: 10.1109/5.58337.

[6] B. Macukow, "Neural networks–state of art, brief history, basic models and architecture," in *Computer Information Systems and Industrial Management: 15th IFIP TC8 International Conference*, Vilnius, Lithuania, Sept. 2016, pp. 1–14. Springer International Publishing.

[7] D. E. Rumelhart, G. E. Hinton, y R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[8] "Tipos de redes neuronales (Clasificación)," [Online]. Available: https://inteligencia-artificial.dev/tipos-redes-neuronales/#:~:text=Los%20tipos%20de%20redes%20neuronales,redes%20neuronales%20de%20base%20radial. [Accessed: Oct. 2, 2024].

[9] "05.7 Redes Neuronales Convoluciones," [Online]. Available: https://dcain.etsin.upm.es/~carlos/bookAA/05.7_RRNN_Convoluciones_CIFAR_10_INFORMATIVO.html. [Accessed: Oct. 4, 2024].

[10] R. M. Schmidt, "Recurrent neural networks (RNNs): A gentle introduction and overview," *arXiv preprint*, arXiv:1912.05911, 2019.

[11] R. C. Staudemeyer y E. R. Morris, "Understanding LSTM--a tutorial into long short-term memory recurrent neural networks," *arXiv preprint*, arXiv:1909.09586, 2019.

[12] "Learn the Basics — PyTorch Tutorials 2.5.0+cu124," [Online]. Available: https://pytorch.org/tutorials/beginner/basics/intro.html. [Accessed: Oct. 7, 2024].

[13] "What is PyTorch?," [Online]. Available: https://www.ibm.com/topics/pytorch. [Accessed: Oct. 7, 2024].

[14] "Building Models with PyTorch," [Online]. Available: https://pytorch.org/tutorials/beginner/introyt/modelsyt_tutorial.html. [Accessed: Oct. 7, 2024].

[15] "Build the Neural Network," [Online]. Available: https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html. [Accessed: Oct. 7, 2024].

[16] “Datasets & DataLoaders,” [Online]. Available: https://pytorch.org/tutorials/beginner/basics/data_tutorial.html. [Accessed: Oct. 8, 2024].

[17] J. Sensio, “Torchvision,” [Online]. Available: https://www.juansensio.com/blog/055_torchvision. [Accessed: Oct. 11, 2024].

[18] “Numpy Introduction,” [Online]. Available: https://www.w3schools.com/python/numpy/numpy_intro.asp. [Accessed: Oct. 4, 2024].

[19] “Matplotlib: Visualization with Python,” [Online]. Available: https://matplotlib.org/. [Accessed: Oct. 4, 2024].

[20] F. Rosenblatt, "Perceptron simulation experiments," *Proceedings of the IRE*, vol. 48, no. 3, pp. 301–309, 1960.

[21] M. Vidal González, "El uso del Perceptrón Multicapa para la clasificación de patrones en conductas adictivas," 2014, pp. 5–9.

[22] M. Belkin et al., "Reconciling modern machine-learning practice and the classical bias–variance trade-off," *Proceedings of the National Academy of Sciences*, vol. 116, no. 32, pp. 15849–15854, 2019.

[23] K. Shukla y N. Fricklas, *Machine Learning with TensorFlow*. Manning Publications, 2018.

[24] P. Ramachandran, B. Zoph, y Q. V. Le, "Searching for activation functions," *arXiv preprint*, arXiv:1710.05941, 2017.

[25] Y. Bai, "ReLU-function and derived function review," in *SHS Web of Conferences*, vol. 20, EDP Sciences, 2022, p. 02006.

[26] D. E. Rumelhart, G. E. Hinton, y R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[27] M. Cilimkovic, *Neural networks and back propagation algorithm*. Institute of Technology Blanchardstown, Dublin, 2015.

[28] P. A. Blanco, "Algoritmo de retropropagación," [Online]. Available: http://www.cs.us.es/~fsancho/ficheros/IAML/2016/Sesion04/seminario_BP.pdf. [Accessed: Oct. 5, 2024].

[29] A. van Ooyen y B. Nienhuis, "Improving the convergence of the back-propagation algorithm," *Neural Networks*, vol. 5, no. 3, pp. 465–471, 1992.

[30] Y. H. Zweiri, J. F. Whidborne, y L. D. Seneviratne, "A three-term backpropagation algorithm," *Neurocomputing*, vol. 50, pp. 305–318, 2003.

[31] “¿Qué es el subajuste?,” [Online]. Available: https://www.ibm.com/es-es/topics/underfitting. [Accessed: Nov. 6, 2024].

[32] “¿Qué es el sobreajuste?,” [Online]. Available: https://www.ibm.com/es-es/topics/overfitting. [Accessed: Nov. 6, 2024].

[33] “¿Cuál es la importancia del tamaño del lote en el entrenamiento de una CNN? ¿Cómo afecta el proceso de formación?,” [Online]. Available: https://es.eitca.org/artificial-intelligence/eitc-ai-dlpp-deep-learning-with-python-and-pytorch/convolution-neural-network-cnn/training-

convnet/examination-review-training-convnet/what-is-the-significance-of-the-batch-size-in-training-a-cnn-how-does-it-affect-the-training-process/. [Accessed: Nov. 6, 2024].

[34] "¿Cuál es la importancia de la tasa de aprendizaje y el número de épocas en el proceso de aprendizaje automático?," [Online]. Available: https://es.eitca.org/inteligencia-artificial/eitc-ai-tff-tensorflow-fundamentos/tensorflowjs/construyendo-una-red-neuronal-para-realizar-la-clasificaci%C3%B3n/examen-revisi%C3%B3n-construcci%C3%B3n-de-una-red-neuronal-para-realizar-la-clasificaci%C3%B3n/%C2%BFCu%C3%A1l-es-la-importancia-de-la-tasa-de-aprendizaje-y-el-n%C3%BAmero-de-%C3%A9pocas-en-el-proceso-de-aprendizaje-autom%C3%A1tico%3F/. [Accessed: Nov. 6, 2024].

[35] "Inicialización de los pesos y bias en Deep Learning," [Online]. Available: https://keepcoding.io/blog/inicializacion-pesos-bias-deep-learning. [Accessed: Nov. 6, 2024].

[36] S. Madhavan y M. T. Jones, "Deep learning architectures," *IBM Developer*, 2017.

[37] A. Gulli y S. Pal, *Deep Learning with Keras*. Packt Publishing, 2017.

[38] J. D. Kelleher, *Deep Learning*. MIT Press, 2019.

[39] Y. Courville, A. Bengio, y I. Bengio, *Deep Learning*, vol. 1. MIT Press, 2016.

[40] R. Pascanu, T. Mikolov, y Y. Bengio, "On the difficulty of training recurrent neural networks," in *International Conference on Machine Learning*, 2013, pp. 1310–1318.

[41] S. Hochreiter y J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: 10.1162/NECO.1997.9.8.1735.

[42] W. S. McCulloch, "On Computable Numbers," *Nature*, vol. 33, no. 5, pp. 22-24.

[43] S.-M. Pedrycz y W. Chen, *Deep Learning: Algorithms and Applications*. Springer, 2020.

[44] "What is LSTM-Long Short-Term Memory," GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/. [Accessed: Nov. 15, 2024].

[45] R. Jaff, "What is LSTM? Introduction to Long Short-Term Memory," Medium, [Online]. Available: https://medium.com/@rebeen.jaff/what-is-lstm-introduction-to-long-short-term-memory-66bd3855b9ce. [Accessed: Nov. 17, 2024].

[46] "The History of Deep Learning — Explored Through 6 Code Snippets," FreeCodeCamp, [Online]. Available: https://www.freecodecamp.org/news/the-history-of-deep-learning-explored-through-6-code-snippets-d0a0e8545202/. [Accessed: Nov. 6, 2024].

[47] "Introduction to machine learning, neural networks and deep learning," Development Seed, [Online]. Available:

https://developmentseed.org/servir-amazonia-ml/docs/Lesson1a_Intro_ML_NN_DL.html. [Accessed: Nov. 6, 2024].

[48] "Tutorial de redes neuronales recurrentes (RNN)," DataCamp, [Online]. Available: https://www.datacamp.com/es/tutorial/tutorial-for-recurrent-neural-network. [Accessed: Dec. 9, 2024].

[49] O. Calzone, "An Intuitive Explanation of LSTM," Medium, [Online]. Available: https://medium.com/@ottaviocalzone/an-intuitive-explanation-of-lstm-a035eb6ab42c. [Accessed: Dec. 4, 2024].

[50] "What is CRISP DM?," Data Science PM, [Online]. Available: https://www.datascience-pm.com/crisp-dm-2/. [Accessed: Jan. 7, 2025].

[51] "CRISP-DM: Fase de 'Comprensión del negocio' (Business Understanding)," MikelNiño, [Online]. Available: https://www.mikelnino.com/2016/11/crisp-dm-metodologia-data-mining-comprension-negocio-business-understanding.html. [Accessed: Jan. 7, 2025].

[52] "Breve historia de las redes neuronales," InteractiveChaos, [Online]. Available: https://interactivechaos.com/es/manual/tutorial-de-machine-learning/breve-historia-de-las-redes-neuronales. [Accessed: Jan. 7, 2025].

[53] "Aprendizaje de Redes Neuronales," La Maquina Oráculo, [Online]. Available: https://lamaquinaoraculo.com/deep-learning/aprendizaje-de-redes-neuronales/. [Accessed: Dec. 2, 2024].

[54] "Optimizador Adam," Ultralytics, [Online]. Available: https://www.ultralytics.com/es/glossary/adam-optimizer. [Accessed: Dec. 2, 2024].

[55] "Explicación de las funciones de pérdida en el machine learning," DataCamp, [Online]. Available: https://www.datacamp.com/es/tutorial/loss-function-in-machine-learning. [Accessed: Dec. 3, 2024].

[56] "Qué es: codificación One-Hot," StatisticsEasily, [Online]. Available: https://es.statisticseasily.com/glossario/what-is-one-hot-encoding/. [Accessed: Dec. 21, 2024].

[57] "Tutorial Pandas: Codificación de variables categóricas a numéricas," Certidevs, [Online]. Available: https://certidevs.com/tutorial-pandas-codificacion-de-variables-categoricas-numericas. [Accessed: Dec. 21, 2024].

[58] “La Función De Pérdida De Huber Y Su Aplicación,” FasterCapital. [Online]. Available: https://fastercapital.com/es/tema/la-funci%C3%B3n-de-p%C3%A9rdida-de-huber-y-su-aplicaci%C3%B3n.html. [Accessed: Jan. 9, 2025].

9 Anexos

9.1 Anexo A

Se deja en este Anexo el informe de originalidad generado por turniting de este presente trabajo de fin de grado (TFG).

InformeOriginalidadTurniting.pdf

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jan 14 19:15:24 CET 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)