



POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL



Graduado en Inginería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Trabajo de fin de grado

Predicción del uso de bicicletas usando redes neuronales recurrentes

Autor: Máximo García Martínez

Director: Antonio García Dopico

MADRID, ENERO, 2021

ÍNDICE

1. Introducción al proyecto	1
1.1. Introducción	1
1.1.1. Bikesharing	1
1.1.2. Big Data	2
1.1.3. Resumen del trabajo	2
1.2. Metodología y desarrollo	3
1.3. Objetivos	3
1.3.1. Objetivos generales	3
1.3.2. Objetivos específicos	3
2. Estado del arte	5
2.1. Inteligencia artifical	5
2.2. Aprendizaje automático	7
2.2.1. Modelos	8
2.3. Redes neuronales cerebrales humanas	9
2.4. Redes neuronales artificiales. Proceso de cálculo hacia delante.	11
2.4.1. Perceptrón	12
2.4.2. Regresión lineal	15
2.4.3. Función de activación	17
2.4.4. Trabajando con matrices	23
2.4.5. Algoritmo de <i>forward-pass</i>	27
2.5. Redes neuronales artificiales. Entrenamiento	30
2.5.1. Entrenamiento de una red	30
2.5.2. Función de coste	33
2.5.3. Minimizando el error	38
2.5.4. Backpropagation	42
2.5.5. Tasa de aprendizaje	48
2.5.6. Optimizadores	49
2.5.7. <i>Overfitting</i>	49
2.6. Redes neuronales recurrentes	54
2.6.1. Nociones básicas	54
2.6.2. Tipos	55
2.6.3. Algoritmo <i>forward pass</i> en RNN	56
2.6.4. Algoritmo <i>backpropagation</i> y descenso del gradiente en RNN	58
2.6.5. Problemas del descenso del gradiente en capas recurrentes "vanilla"	59
2.6.6. Capas LSTM	60
2.7. Trabajos y proyectos similares	62
2.7.1. <i>Development of a station-level demand prediction and visualization tool to support bike-sharing systems' operators</i>	62
2.7.2. <i>Predicting station-level hourly demand in a large-scale bike-sharing network: A graph convolutional neural network approach</i>	63
2.7.3. Competición de Kaggle: <i>Shared bikes demand forecasting</i>	63

3. Desarrollo	65
3.1. Definición del problema	65
3.2. Herramientas y librerías usadas	65
3.3. Preprocesado de datos	66
3.3.1. <i>Feature selection</i> - Unión de <i>datasets</i> y filtro de variables	67
3.3.2. Definición de entradas y salidas de las redes neuronales	70
3.3.3. Modificando el <i>dataset</i> para la red neuronal	72
3.4. Generador de ventanas	75
3.4.1. Código del generador de ventanas	80
4. Modelos usados	81
4.1. Modelo base	81
4.2. Modelos con redes neuronales	82
4.2.1. Modelo denso	82
4.2.2. Modelo recurrente básico	83
4.2.3. Modelo recurrente LSTM	84
4.2.4. Modelo Autoregresivo (AR)	84
4.3. Ventanas	85
4.4. Entrenamiento, validación y test de las redes neuronales	86
5. Resultados	88
5.1. Métricas usadas	88
5.1.1. Pérdida de Huber	88
5.1.2. MAE	88
5.1.3. MSE y RMSE	89
5.1.4. MSLE y RMSLE	89
5.2. Resultados del modelo básico	89
5.3. Redes neuronales	90
6. Problemas encontrados	96
6.1. Estructura del dataset	96
6.2. Desarrollo del preprocesado	96
6.3. Problema de <i>overfitting</i>	97
7. Conclusiones	98
7.1. Estudio de los resultados	98
7.2. Conocimientos adquiridos	98
7.3. Conclusiones en cuanto a la IA	99
8. Trabajo futuro	101
9. Referencias	102
A. Variables matemáticas	107
B. Muestra del <i>dataset</i> original de <i>Divvy</i>	108

C. Muestra del <i>dataset</i> de intervalos	109
D. Código de la clase <i>WindowGenerator</i>	110
E. Código del modelo <i>Auto-Regresivp</i>	112
F. Código usado para compilar, entrenar y evaluar modelos	114

ÍNDICE DE FIGURAS

1.	Representación del funcionamiento de un perceptrón	12
2.	Representación de un perceptrón	13
3.	Representación de un perceptrón	13
4.	Red de perceptrones programada para una puerta lógica XOR	14
5.	Puertas lógicas OR($L_{1,1}$), AND($L_{1,2}$) y XOR($L_{2,1}$)	15
6.	Ejemplo de regresión lineal en un espacio bidimensional	16
7.	Dataset emulando una función seno	18
8.	Comparativa de funciones de activación lineal y no lineal	19
9.	Representación de funcionamiento de una neurona	20
10.	Gráficas de las funciones de activación	22
11.	Red neuronal	24
12.	Proceso de entrenamiento de una red neuronal	32
13.	Error de una regresión para el dato i	33
14.	Visualización del MAE. El error es la media aritmética de todos los errores.	34
15.	Visualización del MSE. El error es la media de las áreas de los cuadrados.	35
16.	Visualización de la pérdida de Huber. Se calcula MSE si $ \hat{y} - y \leq \delta$, en otro caso se calcula MSE.	36
17.	Visualización del MAPE. El error es la media de las proporciones de los errores respecto al valor y	36
18.	Visualización del MPE. Muestra la cantidad de errores que son positivos y la cantidad que son negativos.	37
19.	Proceso iterativo para minimizar el error	39
20.	Ejemplo gráfico de la iteración del descenso del gradiente	40
21.	Representación del descenso del gradiente en una neurona. Lo rojo es la representación del error.	42
22.	Derivadas parciales usadas en el descenso del gradiente.	42
23.	Representación del algoritmo de backpropagation.	43
24.	Red neuronal básica.	44
25.	Ejemplo de <i>overfitting</i> en una regresión lineal.	50
26.	Ejemplo de <i>overfitting</i> en una regresión lineal usando un error como métrica.	51
27.	<i>Dropout</i> aplicado a una red con un 50 % de probabilidades	53
28.	Ejemplo gráfico de un dato único vs. una secuencia.	54
29.	Varios ejemplos de RNN	55
30.	Representación de funcionamiento de los tipos de RNN	56
31.	Funcionamiento de una RNN	56
32.	Estructura de una capa recurrente	57
33.	<i>Backpropagation</i> a través del tiempo en RNN	58
34.	Funcionamiento de una LSTM	61
35.	Mapa interactivo de la red de estaciones de alquiler de bicletas en Chicago [43]	65
36.	Estructura del proyecto dividida por módulos.	67
37.	Estructura del módulo <i>feature-selection</i>	69

38. Ejemplo de red neuronal que usa 3 intervalos de entrada y predice 1 intervalo	71
39. Ejemplo de red neuronal que usa 2 intervalos de entrada y predice 3 intervalos	71
40. Posibles formas de representar la variable <i>hour</i> .	75
41. Ventana con un múltiples entradas y múltiples salidas	77
42. Ventana con un único intervalo de salida	77
43. Ventana con <i>offset</i>	77
44. Predicciones únicas.	78
45. Predicción auto-regresiva.	79
46. Funcionamiento de una ventana deslizante.	79
47. Predicciones del modelo básico	81
48. Modelo denso	82
49. Modelo recurrente básico	83
50. Modelo recurrente LSTM	84
51. Modelo recurrente AR	85
52. Resultados usando pérdida de Huber	93
53. Métricas usando MAE	94
54. Métricas usando MSE	94
55. Métricas usando MSLE	95
56. Métricas usando RMSE	95

ÍNDICE DE TABLAS

1.	Puerta lógica XOR	13
2.	Ejemplo de <i>dataset</i> con viviendas	30
3.	Ejemplo de <i>dataset</i> guardado en <code>trips.csv</code>	72
4.	Ejemplo del <i>dataset</i> agrupados por estaciones y por intervalos.	73
5.	Ejemplo de <i>dataset</i> que contiene los intervalos.	73
6.	Ejemplo de <i>dataset</i> final que será usada por la red neuronal.	74
7.	Métricas obtenidas del modelo básico.	90
8.	Muestras aleatorias del <i>dataset</i> original de Divvy	108
9.	Muestras aleatorias del <i>dataset</i> dividido por intervalos	109

Resumen

Las redes neuronales recurrentes (RNN en inglés) son un tipo de redes neuronales que utilizan conexiones de retroalimentación. Existen diferentes aplicaciones para este tipo de redes neuronales como por ejemplo el estudio de la demanda de un activo durante un tiempo. Este proyecto se llevó a cabo para estudiarlas teóricamente y desarrollar modelos para predecir la demanda de bicicletas en la ciudad de Chicago basados en el enfoque de las RNN y comparar distintas arquitecturas. Los modelos empleados han sido redes neuronales Feedforward, Simple Recurrent Neural Network, LSTM y arquitecturas auto-regresivas. Los modelos predicen en base a una ventana deslizante de intervalos de 1 hora. La salida de los modelos es un vector con valores con la predicción de cada estación de la ciudad para un determinado intervalo. La fecha y la hora es la única entrada que la red neuronal y con ella se han obtenido resultados sobresalientes. Los resultados obtenidos con las redes LSTM y autoregresivas han obtenido resultados que compiten con proyectos de última generación, a pesar de que se han encontrado varios puntos de fallo y posibles mejoras.

Palabras clave: Redes neuronales, redes neuronales recurrentes, predicción, series de tiempo, alquiler de bicicletas.

Abstract

Recurrent neural networks (RNN) are a type of neural network that uses feedback connections. There are different applications for this type of neural network, such as studying the demand for an asset over time. This project was carried out to study them theoretically and develop models to predict the demand for bicycles in the city of Chicago based on the RNN approach and to compare different architectures. The models used were Feedforward Neural Networks, Simple Recurrent Neural Network, LSTM and auto-regressive architectures. The models predict on the basis of a sliding window of 1 hour intervals. The output of the models is a vector with values with the prediction of each station in the city for a given interval. The date and time is the only input that the neural network and with it outstanding results have been calculated. The results obtained with the LSTM and autoregressive networks have obtained results that compete with state-of-the-art projects, despite the fact that several points of failure and possible improvements have been found.

Keywords: Neural network, recurrent neural network, forecasting, time series, bike sharing.

1. INTRODUCCIÓN AL PROYECTO

1.1. Introducción

1.1.1. Bikesharing

Desde mediados del siglo XX, el vehículo privado ha sido el medio de transporte imperante en la mayoría de las ciudades europeas, constituyendo ello, probablemente, el principal factor del diseño urbano hoy en día. Es un transporte rápido y eficaz que ha traído impactos negativos tanto al medio ambiente como a sus habitantes: congestión, contaminación, ruido, gran consumo de energía por persona trasportada, accidentes que suman gran cantidad de años de vida perdidos y discapacidades, contribución al aumento de CO₂... Todo esto está favoreciendo que cada vez sea más atractivo el uso de transporte más sostenible. Estos efectos negativos han provocado preocupaciones y promovido el transporte público, bicicleta o, simplemente, caminar. Asimismo, es sabido que el crecimiento económico de un territorio está estrechamente relacionado con las capacidades de movilidad de éste [1], por lo que, es esencial, que exista un cambio hacia modos de transporte más sostenibles y eficientes con su uso integrados de forma óptima en el devenir de una ciudad. La bicicleta, actualmente, se encuentra en el foco de interés por parte Organismos Públicos por ser el vehículo más sostenible, barato y fácil de implementar como forma de transporte en las ciudades.

En la sociedad del mundo occidental está surgiendo un cambio de paradigma hacia una economía colaborativa donde se permite el acceso a un bien antes que a su propiedad. Es una economía que está prosperando considerablemente esta última década debido al apogeo de las redes sociales, de las tecnologías en tiempo real, de los nuevos patrones de consumo y de las preocupaciones medioambientales. Los *Product Service System* (PSS) son un resultado de este modelo de economía y se basan en que no se ha de poseer un producto para disfrutar de la necesidad que satisface. Los sistemas de movilidad también han evolucionado y ocupado su sitio como PSS y fruto de ello ha sido la aparición de nuevas empresas de *carsharing*, *bikesharing* o *ridesharing*.

En la mayoría de los casos, el negocio de *bikesharing* consiste en colocar numerosas estaciones repartidas por grandes ciudades. El uso de estas bicicletas es simple: Un usuario desea ir de un punto a otro punto de la ciudad, con una app instalada en su móvil alquila una bicicleta en alguna de estas estaciones. Una vez verificado por parte de la empresa de bicicletas que este usuario tiene fondos económicos suficientes, se desbloquea una de las bicicletas. El usuario comenzará a pagar por cada minuto de uso, una tasa que depende de la empresa. O bien, la empresa ofrece un modo de pago en forma de tasa mensual o anual. Una vez que el usuario llega al destino, aparcará la bicicleta en alguna plaza disponible para ello y la bloqueará. Las empresas que gestionan y mantienen estos servicios suelen disponer de herramientas para facilitar el uso a los ciudadanos, en forma de servicios de atención al cliente, páginas webs o aplicaciones móviles que informan de la disponibilidad en tiempo real. No disponen de información sobre la predicción de disponibilidad en un futuro cercano, al menos como servicio al ciudadano. En este aspecto,

es donde este trabajo de fin de grado toma su sentido y su ser.

1.1.2. Big Data

El avance de las tecnologías de información ha generado nuevos requisitos que las herramientas tradicionales son incapaces de proporcionar. Encontrar respuestas con procesos "tradicionales" con la ingente cantidad de datos que hoy en día se trabajan serían procesos muy lentos y caros. Estos problemas se pueden resolver usando nuevos algoritmos basados en la búsqueda de patrones en los datos usando técnicas de *data mining* y *machine learning*. El dueño de los datos puede hacer un análisis y entender más haya allá de lo que las herramientas convencionales creando nuevas oportunidades de negocio que puede aprovechar.

La definición de *Big data* es la de los sistemas de almacenamientos de grandes volúmenes de datos. Uno de los primeros ejemplos del uso de esta tecnología se puede atribuir a *Alan Turing*, cuando en la 2º guerra mundial trabajo en la máquina *Enigma* [2]. Esta máquina consiguió descifrar el código que empleaba el ejército alemán para el cifrado de sus mensajes. Se considera como tal ya que la cantidad de datos para poder cumplir su objetivo fue gigantesca. Desde entonces este campo no ha parado de crecer y su evolución ha estado siempre ligado con la Inteligencia Artificial. Para la detección de patrones comentados anteriormente, hace falta el uso de métodos que la Inteligencia Artificial propone.

Pero no ha sido hasta el nacimiento de Internet y en concreto de las redes sociales cuando ha habido una revolución de los datos. Esto ha provocado la generación de datos masivos que, junto al aumento de la capacidad de cómputo, nuevas industrias han surgido alrededor de esta tecnología. Esta cantidad de datos ha facilitado también que nuevos métodos de Aprendizaje Automático se hayan inventado convirtiendo el campo de la Inteligencia Artificial un campo en auge y que está proponiendo gran cantidad de soluciones.

1.1.3. Resumen del trabajo

Comentado los conceptos de *bikesharing* y *Big data*, el presente trabajo aúna ambas nociones. Este trabajo práctico pretende demostrar las capacidades que tienen las redes neuronales para realizar tareas de predicción y el estudio de su funcionamiento, comportamiento y optimización.

El problema que se quiere resolver es la predicción de uso de bicicletas en las distintas estaciones en una ciudad. Se trata de optimizar el servicio, la satisfacción del cliente y la rentabilidad. Con el análisis de datos se preverá dónde harán falta bicicletas, en estado adecuado para su uso, en un futuro inmediato. Además, muchas de las bicicletas que se alquilan son eléctricas y si estas no son cargadas en la estación, la empresa a de hacerse cargo de ellas llevándolas a un punto de carga y reubicándola en la ciudad. Con la ayuda

de un modelo que prediga donde habrá más demanda en el futuro cercano, se podría ubicar las bicicletas optimizando su oferta.

Para el desarrollo de este proyecto se ha elegido un problema simple que se resolverá con distintas arquitecturas de redes neuronales y se irá mejorando progresivamente añadiendo más conceptos y complejidad en cada iteración. Se estudiará y se comparará distintas arquitecturas de redes neuronales y sus diferentes patrones. Usaremos arquitecturas como *feed-forward*, Red neuronal recurrente (RNN), *Long-Short Term Memory* (LSTM) y Autoregresivo (AR). Se explicará el motivo de uso de cada una de estas redes y los resultados obtenidos.

Se trabajará con un *dataset* facilitado por el ayuntamiento de Chicago y la empresa *Wibee*. Este *dataset* contiene información crucial respecto a todos los viajes que se alquilaron durante los años 2014 y 2019. La ciudad de Chicago cuenta con más de 600 estaciones de esta empresa y fueron más de 26 millones de viajes los que se realizaron durante este periodo.

1.2. Metodología y desarrollo

El presente trabajo consiste en una parte teórica, sobre el estudio de redes neuronales, centrándose en las recurrentes y por otro lado en una parte práctica. Ambas partes irán confluyendo, complementándose y con ayuda del tutor se irán resolviendo dudas o sugerencias de mejora.

1.3. Objetivos

Los objetivos a lograr son los siguientes:

1.3.1. Objetivos generales

- Estudio sobre redes neuronales: Entendimiento de las redes neuronales, su funcionamiento y algoritmos usados en ellas.
- Estudio de las redes neuronales recurrentes como una extensión de las redes neuronales y los desafíos que conlleva.
- Permitir a los responsables de la toma de decisiones del sistema de bicicletas compartidas reequilibrar proactivamente la actuación y asistencia prestada en las estaciones de servicio basadas en predicciones precisas del futuro flujo de bicicletas.

1.3.2. Objetivos específicos

- Redacción del presente documento referenciando todo el proceso llevado a cabo al igual que una explicación de los conceptos aprendidos.

- Uso de librerías de Python 3 de *data mining* como pueden ser: *NumPy* o *Pandas*.
- Uso de librerías de Python 3 de *machine learning* como pueden ser: *Tensorflow* o *Keras*.
- Creación de una red neuronal de tipo *feed-forward* y estudiar su comportamiento con diferentes valores para sus hiperparámetros.
- Creación de una red neuronal recurrente y estudiar su comportamiento con diferentes valores para sus hiperparámetros.
- Creación de una red neuronal LSTM y estudiar su comportamiento con diferentes valores para sus hiperparámetros.
- Comparar resultados de los distintos modelos e intentar justificar su funcionamiento.

2. ESTADO DEL ARTE

2.1. Inteligencia artifical

La Inteligencia Artificial (IA) pertenece al ámbito de las Ciencias de la Computación que durante la última década ha logrado gran reconocimiento y popularidad. A pesar de ser una disciplina que se ha desarrollado en el último lustro prácticamente, ya ha conseguido dar solución a gran cantidad de problemas que en el pasado se consideraban irresolubles o de gran complejidad. Logros como transcripción de voz a texto [3] o viceversa [4], automatización de procesos con robots [5], conducción automática [6], transferencia de estilo [7] son algunos de los ejemplos más famosos. Los problemas que intenta solucionar la IA abarcan a problemas que afectan a cualquier ámbito de la vida.

Dar una definición exacta de lo que es la IA es algo difícil, ya que es un concepto que depende de la propia definición de inteligencia que hoy en día tiene múltiples interpretaciones. Muchos autores proponen su propia definición de IA [8]-[14] y si tomamos todas ellas, podríamos extraer una idea común: la IA es una disciplina de la informática que tiene como objetivo crear entidades que puedan **imitar** comportamientos inteligentes, desde analizar patrones, clasificar elementos en una imagen, predecir valores hasta reconocer voces, conducir, comercializar en bolsa y muchas entidades más en las que un sistema puede simular un comportamiento inteligente.

Principalmente la forma de definir el concepto de IA se puede agrupar en alguno de los siguientes enfoques [15]:

1. Sistemas que piensan como humanos [8], [9]: Las capacidades del sistema deben ser propias de seres humanos. Alan Turing (1950) propuso la Prueba de Turing [16], la cual pretende poner a prueba estas capacidades y de ser superada, quedaría demostrado que el sistema tendría un comportamiento similar al de un ser humano. La fiabilidad de esta prueba es cuestionable dado que es fácil crear un sistema que imite el comportamiento humano. Imitar no es tener inteligencia propia. [15]
2. Sistemas que actúan como humanos [12], [13]: La prueba de Turing no tenía en cuenta la persona física sino únicamente su inteligencia. Hanard (1991) propuso la prueba total de Turing[17] que tiene en cuenta todo tipo de estímulos externos, habilidades de percepción y manipulación de objetos. O sea, añade nuevos requisitos al sistema: visión por computador y robótica que precisan nuevas capacidades: robótica, procesamiento de lenguaje natural, representación del conocimiento, razonamiento automatizado y aprendizaje automático. Estas seis disciplinas son seis de las disciplinas de la IA que en la actualidad son objeto de estudio como se explicará más adelante. [15]
3. Sistemas que piensan racionalmente [10], [11]: El sistema trataría de imitar la estructura y procesado de la información a como lo haría un ser humano. Para ello, habría que investigar y tener mayor desarrollo de teoría precisa sobre la mente, para así poder plasmar dicha teoría en un sistema. Esto se podría hacer o bien mediante

un estudio introspectivo individual o mediante experimentos psicológicos que sirvan para realizar distintas definiciones y requisitos para que el sistema se pueda programar. Para llevar a cabo una prueba, se facilitarían los mismos datos con una estructura y se medirían los tiempos de reacción. Si la estructura y datos de salida, junto al tiempo de reacción son similares a los de un humano, existe evidencia que algunos de los mecanismos del programa se pueden comparar con los que utiliza un ser humano y, por tanto, considerarse inteligente. [15]

4. Sistemas que actúan racionalmente [14], [18]: Ya no serían sistemas los que se construirían sino agentes. Un agente es algo que razona (agente viene del latín *age-re*, hacer). De este agente se espera que actúe con la intención de alcanzar el mejor resultado incluso pudiendo tomar la decisión de inacción, si lo considera oportuno en situaciones de incertidumbre.

Se considera que este agente debe ser capaz de realizar inferencias a la hora de establecer qué acciones ayudarán a la consecución del objetivo buscado. Para obtener una correcta inferencia no solo depende de la racionalidad, ya que, en ocasiones, la mejor decisión no será consecuencia de un lento proceso de deliberación sino de un acto reflejo o incluso situaciones que no hay nada correcto que hacer y en las que hay que tomar una decisión. Este enfoque, es el más general y complejo a nivel técnico, porque propone el desarrollo de distintas partes: una parte racional que es común al resto de enfoques y una parte de inferencia lógica. Por otro lado, es más sencillo llevar a cabo pruebas con estos agentes ya que la racionalidad está bien definida matemáticamente, y, por lo tanto, se puede descomponer para diseñar agentes que puedan lograrlo de forma comprobable. [15]

Actualmente, las inteligencias artificiales creadas son lo que se conocen de tipo débil. Estos, son sistemas que pueden superar las capacidades de un humano en alguna tarea. Ahora bien, si se selecciona uno de estos sistemas que sobresalen en un dominio muy específico y se intenta que realice otra tarea, el resultado de esta será mucho peor que el de un ser humano. Esta capacidad de poder hacer múltiples tareas al mismo tiempo es una característica muy codiciada en la actualidad que se sigue investigando en todos los departamentos de IA. Por el momento, solo existen Inteligencias Artificiales débiles, capaces de hacer extraordinariamente una o varios grupos de tareas. Por el contrario, los sistemas con IA fuertes son sistemas que pueden llevar a cabo tareas de distintos dominios, pero aún no existe ejemplo de este tipo de sistemas [15].

Es importante mencionar que algunos de los cuatro enfoques sobre la definición de IA explicados anteriormente tienen como objetivo imitar comportamientos inteligentes. Tratar de imitar un comportamiento puede ser relativamente fácil. Por ejemplo, se puede programar a un sistema para que juegue al ajedrez siguiendo una serie de reglas y predicados. Al ser una máquina podrá haber memorizado partidas similares que hubo en el pasado y saber cuáles fueron las estrategias que tomó su contrincante y de ese modo tener acceso a mucha más información que su rival. De ese modo, el sistema podrá escoger la mejor decisión en cada movimiento y será imbatible, pero si tratamos de que intente

jugar con alguna estrategia que no haya visto antes, no será capaz de usar una estrategia que tenga válida tácticamente hablando, es decir, no funcionará para otro casos porque el sistema solo está imitando y no aprendiendo.

Imitar no significa que dicho comportamiento sea en esencia un comportamiento cognitivo, sin embargo, según la definición que se ha dado de IA, el sistema que juega al ajedrez se consideraría un sistema con IA, aunque las mecánicas del movimiento se haya memorizado. Es por ello por lo que dentro de este campo de la informática podemos encontrarnos diferentes subcategorías que responden a diferentes comportamientos inteligentes. Estas subcategorías son:

- Robótica
- Visión por computador
- Procesamiento de lenguaje natural
- Representación del conocimiento
- Razonamiento automático
- Aprendizaje automático

Por encima de todo, si hay una categoría que nos define como agentes inteligentes es la capacidad de aprender. Esta capacidad es la que se centra este trabajo y es objeto de estudio del aprendizaje automático que vamos a ver en más detalle a continuación.

2.2. Aprendizaje automático

El aprendizaje automático (*Machine Learning* (ML) en inglés) es la rama de la IA que estudia como dotar a las máquinas de capacidad de aprendizaje entendiendo a éste como la generalización del conocimiento a partir de un conjunto de experiencias. Este aprendizaje puede dividirse en [15]:

- Supervisado: El algoritmo aprende en un conjunto de datos etiquetados, proporcionando una clave de respuesta que el algoritmo puede utilizar para evaluar su precisión en los datos de entrenamiento [19].
- No supervisado: Dados los datos no etiquetados, el algoritmo trata de captar sentido extrayendo características y patrones por sí mismo
- Reforzado: Se entrena un algoritmo con un sistema de recompensas, proporcionando retroalimentación cuando un agente de inteligencia artificial realiza la mejor acción en una situación particular [19].

El aprendizaje automático es la rama principal de la IA, ya que el resto de las categorías o bien imitan comportamientos o bien aprenden de experiencias, es decir, usan *Machine Learning* (ML) para realizar la tarea que se les encomienda. Una cosa es programar una máquina para que pueda moverse y otra cosa es que la máquina aprenda a moverse. Igualmente, no es lo mismo programar que componentes forman una cara de una persona que automáticamente aprender que es una cara. Este cambio de paradigma es lo que diferencia el ML de la IA, y es por ello por lo que no se debe pensar que son los mismos conceptos.

El aprendizaje automático es el estudio sistemático de algoritmos y sistemas que mejoran su conocimiento o desempeño con experiencia. En esta subcategoría existen diferentes tipos de aplicaciones siendo algunos de ellos [15]:

- Modelos de regresión: Son modelos que predicen el valor de una o más variables dado un vector de entrada. Una función lineal es el modelo de regresión más simple, pero los modelos de regresión más complejos están formados por un conjunto fijo de funciones no lineales, conocidas como funciones base [20]. Algoritmos como Support Vector Machine (SVM) o las Red neuronal (NN) se basan en este tipo de modelos. Este último, será el usado en el presente trabajo.
- Árboles de decisión
- Modelos de clasificación
- Técnicas de agrupación

2.2.1. Modelos

El universo que se conoce está en constante evolución y es complejo, caótico y enigmático, sin embargo, la inteligencia del ser humano consigue dar sentido a todo ese caos en una búsqueda por la elegancia y la simetría que se esconde entre los patrones que identifica en nuestra realidad. La habilidad de detectar patrones y usarlo para nuestro bien ha sido una de las principales razones para el desarrollo de la especie humana. La ciencia nos ha permitido entender, observar y simplificar el mundo convirtiendo todo este enigma en conocimiento, es decir, reconstruyendo la realidad a través de modelos [21].

Un modelo es una construcción conceptual y simplificada de una realidad compleja permitiendo entender mejor dicha realidad. Existen multitud de modelos que usamos diariamente, por ejemplo, un mapa. Un mapa nos permite reflejar un mundo tridimensional en una superficie bidimensional eliminando información que no necesitamos procesar como puede ser artefactos del entorno o tipo de vegetación. Otro ejemplo, es una ecuación física, donde se relacionan distintas constantes y valores y de ese modo poder aproximar el comportamiento físico de la realidad. Una partitura también es otro ejemplo de modelo. En este se refleja información de como distintos instrumentos deben coordinarse para poder producir siempre la misma canción. Se podría usar el espectro de frecuencias de la canción para poder representar mejor la realidad, pero esto obviamente, es mucho más

complejo de interpretar siendo un ser humano. En resumen, un modelo busca el equilibrio entre representar correctamente la realidad y ser simple para que pueda ser usado [22].

Imaginemos que se quiere modelar la meteorología. Para ello se recopilan diferentes evidencias y tras observar se pueden enunciar un primer modelo:

“El verano será soleado, caluroso y despejado. El invierno será frío, nublado y lloverá.”

Si se sigue recopilando evidencias pronto uno se dará cuenta de que este modelo es muy simple ya que habrá días que hará frío en verano y calor en invierno. Habrá tormentas en verano y días despejados en invierno. Esto provocará mejoras en el modelo en cada iteración.

Pero si se sigue estudiando otras evidencias en algún trópico o en algún polo, por ejemplo, se llegará a la conclusión de que el modelo era muy simple y por lo tanto, se tendrá que añadir más reglas para que este modelo se asemeje mejor a la realidad en cualquier parte del mundo.

Al final, se obtendrá un modelo muy complejo con todas las excepciones y condiciones. Una alternativa a esto es hacer uso de la probabilidad para poder decir matemáticamente que la mayoría de las veces un día de verano será soleado y no un modelo tan complejo del que depender.

La probabilidad es la herramienta perfecta para acotar la incertidumbre sobre un tema por falta de conocimientos o datos o para evitar el trabajo de realizar un modelo complejo que conllevaría menor comprensión para la mente humana y dispersión en su aproximación. Poder usar una probabilidad es mucho más fácil que tener que estudiar todas las condiciones físicas de un entorno y el comportamiento de sus entidades para poder saber a ciencia cierta lo que va a ocurrir. Utilizar la probabilidad para construir modelos da como resultado los modelos probabilísticos. Estos modelos comprimen en base a probabilidades muchas de la variabilidad de nuestra realidad siendo más sencillo de gestionar la información que recibimos del entorno [22].

Nuestro cerebro aplica esquemas similares a estos modelos probabilísticos y gracias a ellos es que tenemos capacidades como la de conceptualizar, predecir, generalizar, razonar o aprender. Por esto mismo, descubrir cuales son estos modelos es uno de los objetivos básicos del campo del *Machine Learning* y una de las herramientas fundamentales de la IA.

2.3. Redes neuronales cerebrales humanas

Las redes neuronales "artificiales" están inspiradas en el cerebro orgánico de los seres humanos llevado a los ordenadores. No es una perfecta comparación, pero donde más similitudes hay entre una red neuronal del cerebro y una artificial es en las neuronas que lo

forman[23]. Las neuronas son las partes más simples de la red neuronal y juntando varias de ellas se pueden formar las redes. En los siguientes párrafos se verá el funcionamiento básico de una neurona y alguno de los procesos del cerebro humano para que se pueda comparar con el funcionamiento de una red neuronal artificial.

Las neuronas son un tipo específico de célula encargadas de enviar información usando señales eléctricas denominadas impulsos nerviosos a otras neuronas o a órganos efectores. Su estudio se remonta a 1888, cuando Ramón y Cajal comenzó a postular una teoría conocida como "la doctrina de la neurona" [24]. En ella se destaca la concepción de la neurona como una unidad discreta y la ley de la polarización dinámica, modelo capaz de explicar la transmisión unidireccional del impulso nervioso. Posteriormente, publicó múltiples múltiples trabajos, entre otras aportaciones describió la hendidura sináptica, sugiriendo que la comunicación entre neuronas se hacía mediante moléculas químicamente bien definidas, llamadas neurotransmisores, sirva como ejemplo una de las moléculas más conocidas, la acetilcolina[25]. Los seres vivos son capaces de reaccionar a las modificaciones del medio. En los seres pluricelulares la captación de estos cambios y la respuesta o falta de la misma es la función del tejido nervioso, formado por neuronas. El ejemplo más fácilmente "entendible" es la respuesta ante un estímulo sea un movimiento producido por una contracción muscular.[26]

Las tareas de una neurona son:

- Conducción y trasmisión de impulsos nerviosos. El mecanismo más simple de acción nerviosa está representado por el arco reflejo monosináptico, que consiste en un circuito neuronal formado por dos neuronas: Una neurona sensorial posee un receptor en una terminación para recibir el estímulo (en las neuronas más "clásicas" está en las dendritas). La información se propaga a lo largo del axón mediante variaciones transitorias del potencial de reposo. Esto es, un cambio del potencial eléctrico transmembrana celular que, produce una onda de excitación que se inicia en el extremo que percibe y se agota en el final del axón con la liberación del neurotransmisor a la hendidura sináptica. Este neurotransmisor será el que provoque otro potencial eléctrico en la siguiente neurona (que lo captará en sus dendritas) o un efecto si es un órgano efector el que está al otro lado de la hendidura. El potencial eléctrico que se desencadena es del tipo todo o nada en una neurona. La gradación de más o menos intensidad de una respuesta dependerá del reclutamiento de neuronas conseguido por la intensidad y tipo de estímulo y la trasmisión de la información a otras neuronas dependerá de la cantidad de neurotransmisor liberado que estimulará a una o a muchas neuronas (o a ninguna, una o muchas células efectoras)[27].
- Almacenar información intuitiva y adquirida. La adaptación a las funciones especializadas se efectúa por medio de diferentes tipos de prolongaciones [27]. A diferencia de la memoria que hoy en día tenemos en discos duros, la memoria en el cerebro no se puede guardar como tal en elementos tangibles, es decir, en las neuronas no se guardan los recuerdos. Las memorias son señales. Una memoria a

corto plazo es un patrón de señales en el cerebro que ocurren en unas neuronas específicas en el córtex prefrontal. Esa señal comienza como un potencial eléctrico creado por los iones entrando y saliendo de las células en un extremo de la neurona creando una señal eléctrica la cual desencadena una liberación de sustancias químicas que producirá en la siguiente neurona un cambio de su potencial eléctrico en su membrana celular, la trasferencia de información es liberación de una molécula que produce un cambio eléctrico en la siguiente neurona. Básicamente, la electricidad y la química son las responsables de que una señal sea transportada a través de muchas neuronas [28].

En resumen, una memoria es una reacción en cadena y como se ha mencionado antes, una neurona está conectada como mínimo a miles de neuronas. Por lo que la cantidad de combinaciones y patrones únicos que se pueden generar a partir de las conexiones entre neuronas son casi infinitas[28].

Para que se cree un nuevo recuerdo, primero debe de existir un estímulo. Esos estímulos son captados por alguna de las neuronas y envían esa señal a otras neuronas con un patrón en específico. Esa memoria recién creada es totalmente única y, por lo tanto, esa memoria es el conjunto de unas neuronas activadas con una actividad específica entre ellas. En el futuro, si ese patrón vuelve a volver a activarse producirá que vuelva la memoria en forma de recuerdos a la persona. Si se reactiva una y otra vez (o hay estímulos acompañantes, importantes para el sujeto receptor, entre otros detalles) el patrón será almacenado en el área de memoria a largo plazo en el hipocampo. En esta área, las neuronas están más juntas y la señal entre ellas es más fuerte para que los químicos y las señales eléctricas no tengan que viajar tan lejos y con el tiempo esas memorias se arraigan debido a que las memorias se reactivan con mayor frecuencia. Por otro lado, si un recuerdo no se reactiva dado un periodo de tiempo puede que se desvanezca o se matice o se cambie en nuevos recuerdos. [28].

2.4. Redes neuronales artificiales. Proceso de cálculo hacia delante.

Las redes neuronales son algoritmos que se usan en el campo de la Inteligencia Artificial y tratan solventar problemas en los cuales se trabaja con gran cantidad de datos tratando de buscar patrones en ellos. Además, es una de las pocas alternativas a otros algoritmos que nos son capaces de tratar gran volumen de datos.

Los inicios de las redes neuronales se remontan a la década de 1950, cuando McCulloch y Walter Pitts[29] trabajaron en un modelo matemático que se asemejaba al comportamiento que conocían de una neurona. El científico Frank Rosenblatt, inspirado en este trabajo, desarrolló lo que se conoce como redes de perceptrones. Este fue el primer acercamiento a lo que hoy se conoce como redes neuronales[30].

2.4.1. Perceptrón

Un perceptrón, o neurona es una unidad básica de inferencia en forma de discriminador lineal, es la base de lo que se conoce hoy como neurona artificial[30]. Básicamente, un perceptrón toma varios valores binarios como entrada x_1, x_2, \dots, x_n y produce un único valor de salida y .

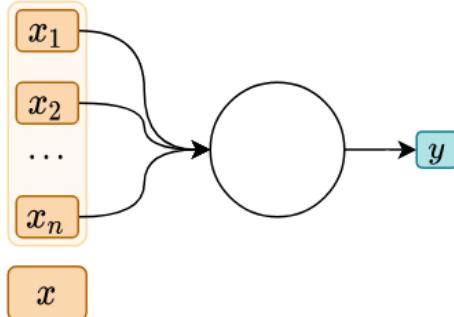


Figura 1: Representación del funcionamiento de un perceptrón

Además del vector x y el valor y , el perceptron tiene dos componentes más:

- El vector de pesos w que expresa la importancia de las respectivas entradas para la salida y tendrá el mismo número de elementos que x . Este vector se usará para calcular la suma ponderada $\sum_i w_i x_i$.
- El umbral o sesgo ($bias(b)$ en inglés): El valor del sumatorio se comprobará si es menor o mayor a este valor b . En función de ello, el valor y será 0 o 1. El umbral es un valor que representa lo fácil que es producir un valor 1 en y . Para un perceptrón con un sesgo muy grande, es extremadamente fácil que el perceptrón produzca un 1. Pero si el sesgo es muy negativo, entonces es difícil que el perceptrón produzca un 1.

Tanto el vector w y el valor umbral b son parámetros que se deben de saber previamente. El valor y producido por el perceptrón viene dado por la siguiente ecuación:

$$y = \begin{cases} 0 & \text{si } \sum_i w_i x_i \leq b \\ 1 & \text{si } \sum_i w_i x_i > b \end{cases} \quad (1)$$

Visto gráficamente:

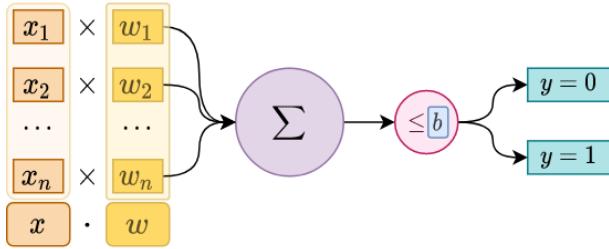


Figura 2: Representación de un perceptrón

Simplificando la ecuación 1 usando el producto vectorial y cambiando de lado b :

$$y = \begin{cases} 0 & \text{si } w \cdot x + b \leq 0 \\ 1 & \text{si } w \cdot x + b > 0 \end{cases} \quad (2)$$

Visto gráficamente:

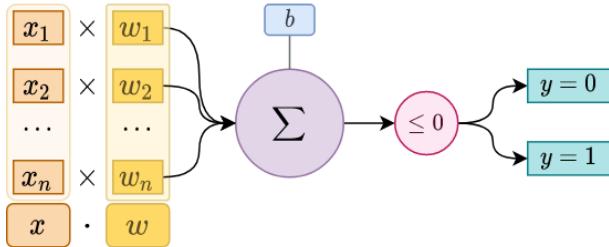


Figura 3: Representación de un perceptrón

Los perceptrones se pueden agrupar formando capas y estas capas se pueden agrupar a su vez formando una red. Se puede ver el funcionamiento de una red neuronal programando una puerta lógica XOR que recibe dos argumentos binarios de entrada y emite una salida como se puede ver a continuación:

A	B	Salida
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 1: Puerta lógica XOR

Una red neuronal que actúa como una puerta XOR es la siguiente:

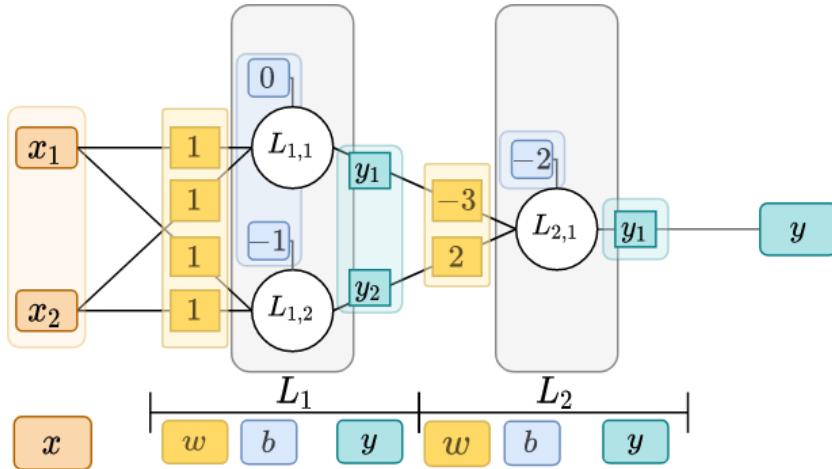


Figura 4: Red de perceptrones programada para una puerta lógica XOR

Las redes se suelen representar con un grafo ponderado unidireccional. El grafo está dividido por capas, en este caso en dos capas (L_1 y L_2). Cada capa tiene como mínimo una neurona, que será representada por $L_{l,i}$ siendo l el número de la capa e i el índice de la neurona dentro de la capa. Una red puede tener tantas capas como se quiera.

A la primera capa se le denomina capa de entrada. A la última capa, capa de salida. Estas dos capas son las que indican el tamaño del vector de entrada y el tamaño del vector de salida que producen. Si una red esta formada por una sola capa, esa misma capa será la que se denominé de entrada y salida. Por otro lado, si una red neuronal tiene más de dos capas, todas las capas entre la capa de entrada y la capa de salida son las que se denominan capas ocultas y el autoajuste que se realiza mediante el algoritmo de *backpropagation* (ver Sección 2.5.4) es difícil de entender y explicar y por eso a las redes neuronales a veces se les denomina cajas negras.

Los valores de los pesos se representan como si fuesen los pesos de las aristas. La cantidad de pesos que tendrá una capa asociada a ella viene dada por la multiplicación del número de neuronas en la capa L_l por el número de neuronas en la capa anterior L_{l-1} . Por último, cada neurona tendrá asociado un valor b que será representado como un término independiente conectado a la neurona por una línea. Por ejemplo, la neurona $L_{2,1}$ tiene asociado el vector $w = [3, -2]$ y el valor $b = -2$.

Con el grafo ya explicado, se resuelve a continuación el problema que se había planteado: Programar una puerta lógica XOR con una red neuronal. Esta red recibe dos valores binarios de entrada y devolverá un único valor binario. A modo de ejemplo se

realizan los cálculos para distintos ejemplos:

$$\begin{aligned} \text{Para } x = (0, 0) \\ y^{L_{1,1}} &\implies (1 \ 1) \cdot x^T + 0 = 0 \implies y^{L_{1,1}} = 0 \\ y^{L_{1,2}} &\implies (1 \ 1) \cdot x^T - 1 = -1 \implies y^{L_{1,2}} = 0 \\ y = y^{L_{2,1}} &\implies (3 \ -2) \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} - 2 = -2 \implies [y = 0] \end{aligned}$$

$$\begin{aligned} \text{Para } x = (0, 1) \\ y^{L_{1,1}} &\implies (1 \ 1) \cdot x^T + 0 = 1 \implies y^{L_{1,1}} = 1 \\ y^{L_{1,2}} &\implies (1 \ 1) \cdot x^T - 1 = 0 \implies y^{L_{1,2}} = 0 \\ y = y^{L_{2,1}} &\implies (3 \ -2) \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} - 2 = 1 \implies [y = 1] \end{aligned}$$

$$\begin{aligned} \text{Para } x = (1, 1) \\ y^{L_{1,1}} &\implies (1 \ 1) \cdot x^T + 0 = 2 \implies y^{L_{1,1}} = 1 \\ y^{L_{1,2}} &\implies (1 \ 1) \cdot x^T - 1 = 1 \implies y^{L_{1,2}} = 1 \\ y = y^{L_{2,1}} &\implies (3 \ -2) \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} - 2 = -1 \implies [y = 0] \end{aligned}$$

2.4.2. Regresión lineal

En realidad, la arquitectura de la red en la figura 4 usa una neurona que tiene como objetivo simular una puerta lógica OR ($L_{1,1}$) y otra neurona una puerta lógica AND ($L_{1,2}$). Viendo la lógica de una puerta XOR:

$$A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B) = (A \vee B) \wedge (\neg A \vee \neg B) \quad (3)$$

Tiene sentido que se usen las puertas OR y AND en la red. Se puede ver gráficamente el funcionamiento de cada neurona en la siguiente imagen:

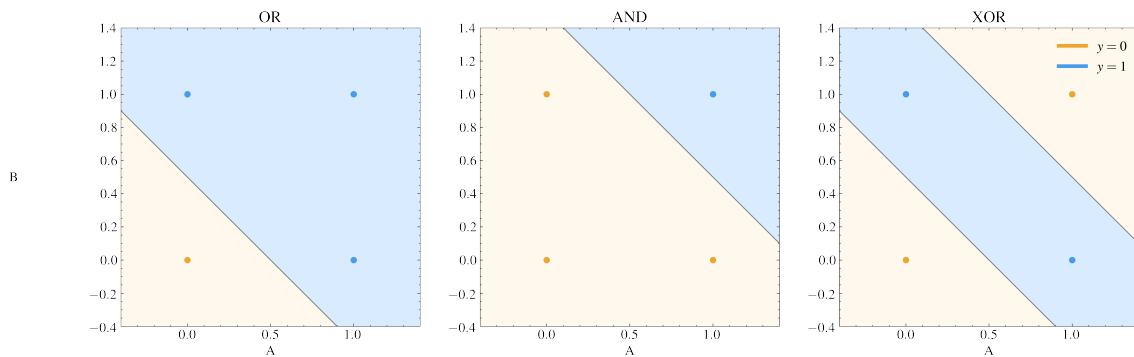


Figura 5: Puertas lógicas OR($L_{1,1}$), AND($L_{1,2}$) y XOR($L_{2,1}$)

Cada una de estas neuronas está realizando una tarea de clasificación, simplemente comprueba si los datos de entrada se encuentran en un lado o en otro de la recta. Esto es así porque se ha definido el perceptrón de tal forma que solo pueda devolver 0 o 1. Pero eliminando ese paso de clasificación que se ha definido en la ecuación 2, se obtendría una regresión lineal tal que:

$$y = w \cdot x + b \quad (4)$$

Una regresión lineal es el otro tipo de tarea que una neurona puede resolver. Esta es una de las principales diferencias entre las neuronas modernas y los perceptrones. Los perceptrones solo están programados para realizar una tarea de clasificación, pero las neuronas se pueden programar para otro tipo de casos. Aunque como se explicará posteriormente, un perceptrón es un tipo de neurona con una función de activación escalonada (ver apartado 2.4.3).

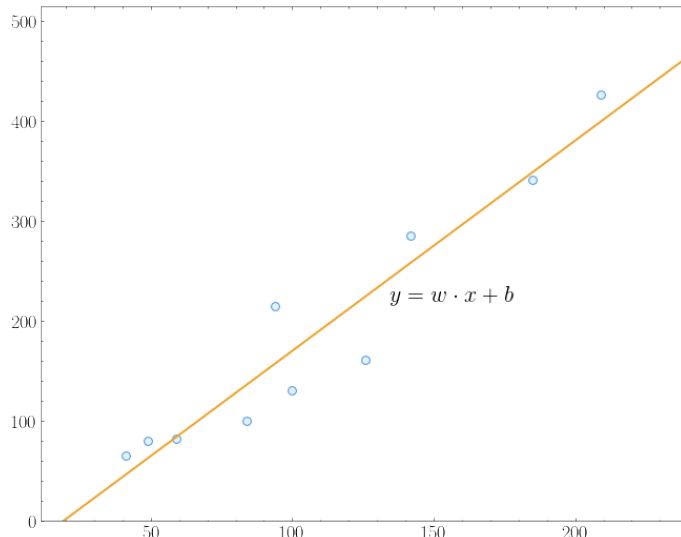


Figura 6: Ejemplo de regresión lineal en un espacio bidimensional

Una regresión lineal es un método que estudia la relación existente entre varias variables y de ese modo genera un modelo que puede ser usado para estimar otros valores. Matemáticamente, en un espacio multidimensional n , una regresión se define de la siguiente manera:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n \quad (5)$$

En esta ecuación hay un término independiente w_0 , y por cada dimensión, habrá un valor w y un valor x asociado a ella. Geométricamente, en un espacio de dos dimensiones, z será una recta, un plano si son tres dimensiones y un hyperplano para mayores de tres dimensiones, que gráficamente no se puede representar. De hecho, se puede observar que la ecuación 4 es en realidad la función de una recta, siendo b el término independiente.

A partir de este punto se usará z para referirse a la regresión lineal que es calculada en una neurona:

$$z \equiv y = w \cdot x + b \quad (6)$$

2.4.3. Función de activación

Lo anterior, se puede comprobar matemáticamente que el efecto de sumar muchas operaciones de regresión lineal equivale a una única regresión lineal. Esto se demuestra en la sección 2.4.5. El perceptrón que se ha creado puede colapsar hasta ser equivalente a tener una única neurona. Para evitar esto, debemos calcular valores que no den como resultado una función lineal(una línea recta en dos dimensiones), necesitamos que cada una de estas neuronas aplique alguna función para realizar algún tipo de manipulación no lineal que distorsione sus valores de salida y para ello se usan las funciones de activación [30].

A modo de ejemplo, se mostrará con un ejemplo con un modelo con solo funciones lineales comparado con un modelo que usa funciones no lineales. Para ello, se expone el siguiente ejemplo: se quiere crear un modelo de regresión que trate de predecir el valor de una función seno. Al ser el seno una función no lineal, se necesitará de funciones no lineales para resolver el problema.

$$y = \sin(x) \quad (7)$$

También se mostrará código para ir introduciendo el funcionamiento de las librerías que se usan popularmente en mundo del *machine learnig*. Como se explica en la sección 3.2, en este trabajo se han usado keras(redes neuronales), numpy(vectores y matrices) y matplotlib(representar funciones) entre otras.

Como se explica en la sección de entrenamiento(sección 2.5.1), se necesita un dataset para que la red se pueda entrenar. El dataset se creará de la siguiente forma:

```
# real sen data
x = np.linspace(0, 2*np.pi, 1000)
y = np.sin(x)

# some noise is added to the real dataset to make it
# look like a real world data set
noise = 0.1
noisex = x + np.random.normal(-noise, noise, 1000)
noisey = y + np.random.normal(-noise, noise, 1000)
```

Gráficamente quedaría una serie de puntos esparcidos alrededor de la función seno como es de esperar:

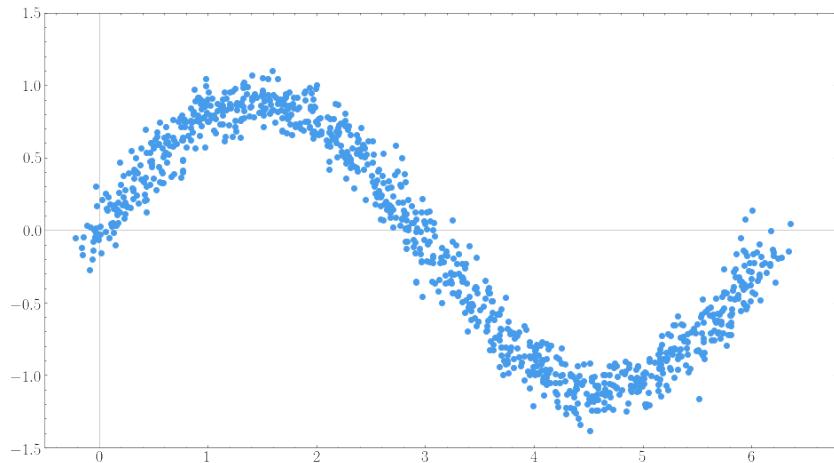


Figura 7: Dataset emulando una función seno

Se crearán dos modelos, es decir, dos redes con arquitecturas distintas: Un modelo con funciones de activación lineales(`linear_model`) y otro con funciones de activación no lineales(`relu_model`).

El primer modelo viene definido de la siguiente forma:

```
linear_model = Sequential()
linear_model.add(Dense(32, activation='linear'))
linear_model.add(Dense(32, activation='linear'))
linear_model.add(Dense(1))
```

Lo que se hace en ese código es crear la arquitectura de la red. Será una red con tres capas densas. Las dos primeras capas son las capas ocultas del modelo y se indica que se quieren 32 neuronas en cada capa con la función de activación `linear`. La última capa simplemente es para que el modelo solo devuelva un único valor.

La otra red usará una función no lineal de activación ReLU, explicada posteriormente. EL código para la arquitectura de esta red es la siguiente:

```
relu_model = Sequential()
relu_model.add(Dense(32, activation='relu'))
relu_model.add(Dense(32, activation='relu'))
relu_model.add(Dense(1))
```

Posteriormente se "compila" el modelo. En este paso, se indicará a *keras* que se parámetros tanto obligatorios como opcionales se quieren para red. Estos parámetros serán explicados en próximas secciones. Justo después se entrena a los modelos.

```
# Compile linear model
linear_model.compile(loss='mean_squared_error',
                      optimizer='adam')
# Train linear model
linear_model.fit(noiseX, noiseY, epochs=100, verbose=0)
```

```

# Compile ReLU model
relu_model.compile(loss='mean_squared_error',
                    optimizer='adam')
# Train ReLU model
relu_model.fit(noisex, noisey, epochs=100, verbose=0)

```

Para realizar una predicción con estos modelos simplemente se puede usar la función `predict()` del modelo:

```

predicted_by_linear = linear_model.predict(x)
predicted_by_relu = relu_model.predict(x)

# Plot predictions
plt.plot(x, predicted_by_linear)
plt.plot(x, predicted_by_relu)

```

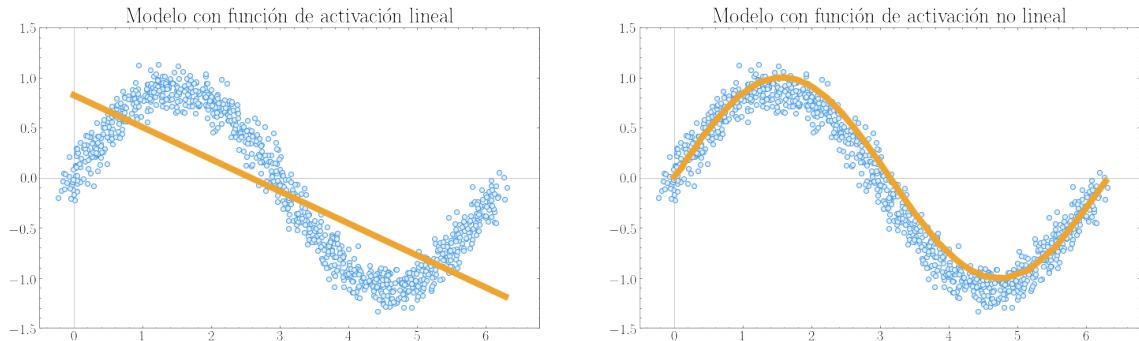


Figura 8: Comparativa de funciones de activación lineal y no lineal

Como se puede observar, el modelo que usa una función de activación lineal se reduce a una única recta mientras que el modelo que usa una función de activación no lineal se adapta correctamente.

En resumen, la función de activación es una función que se aplica al resultado de la suma ponderada de los valores de entrada, es decir, a z . El objetivo es que distorsione el resultado de la neurona añadiéndole deformaciones no lineales para que así se pueda encadenar de forma efectiva la computación de varias neuronas. Se representará esta función de activación de la siguiente forma:

$$a(z) = a(w \cdot x + b) \quad (8)$$

Actualizando la imagen que se ha usado antes para definir un perceptrón, quedaría tal que:

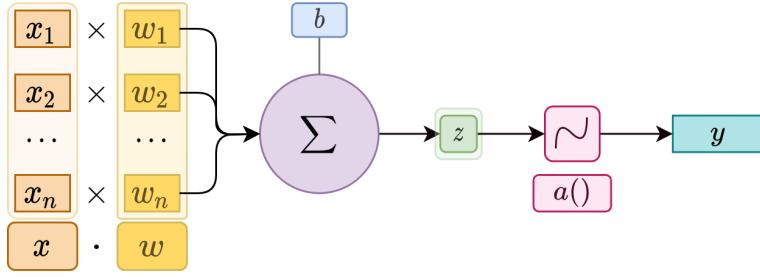


Figura 9: Representación de funcionamiento de una neurona

Existen varios tipos de función de activación, estas son las más usadas:

- Lineal: Es simplemente la ecuación de una recta y por lo tanto una función lineal. Normalmente se utiliza en la última capa en un modelo de regresión, es decir, un modelo que realiza una tarea de regresión y no una tarea de clasificación.

$$a(z) = z \quad (9)$$

$$a'(z) = 1 \quad (10)$$

- Escalonada (*step* en inglés): El objetivo de la función es convertir los valores a 0 o a 1 en función de b . El propósito de esta función de activación es imitar una neurona que "se activa" o "no se activa" basándose en la información de entrada. Tradicionalmente, esta función se usaba en las redes de perceptrones y de hecho, es la usada en la ecuación 1. La ecuación es la siguiente:

$$a(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases} \quad (11)$$

$$a'(z) = 0 \quad (12)$$

- Rectificador de Unidad Lineal (ReLU): Es una función lineal cuando es positiva y constante a 0 cuando el valor es negativo. Es muy usada porque es una función no lineal y parecida a la función lineal, por lo que es rápido de computar.

$$a(z) = \max(0, z) \quad (13)$$

$$a'(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases} \quad (14)$$

Existe una variante de esta función de activación (conocida como *leaky*). En esta variante, se cambia la inclinación a un valor s de la parte negativa respecto al eje de abscisas.

$$a(z) = \begin{cases} s & \text{si } sz \leq 0 \\ z & \text{si } z > 0 \end{cases} \quad (15)$$

$$a'(z) = \begin{cases} s & \text{si } sz \leq 0 \\ 1 & \text{si } z > 0 \end{cases} \quad (16)$$

- Sigmoide: Es la función más usada en las en las redes neuronales. La distorsión que produce a los valores muy grandes hace que se saturen en 1 y los valores muy pequeños hace que se saturen a 0. Esta función es muy útil para representar probabilidades ya que siempre vienen en el rango de $[0, 1]$ y como se explicó en el apartado 2.2.1, la probabilidad es la herramienta perfecta para el *machine learning*. Además, esta función, junto con \tanh , se consigue una fluidez que no se consigue con las funciones anteriormente explicadas. Un pequeño cambio en w o en b producirá un pequeño cambio en el valor de salida.

$$a(z) = \frac{1}{1 + e^{-z}} \quad (17)$$

$$a'(z) = a(z)(1 - a(z)) \quad (18)$$

- Tangente hyperbolica (\tanh): Es similar a la sigmoide, pero el rango de los valores de salida es de $[-1, 1]$.

$$a(z) = \tanh(z) \quad (19)$$

$$a'(z) = 1 - \tanh(z)^2 \quad (20)$$

A continuación, se muestra gráficamente cada una de las funciones explicadas de forma gráfica:

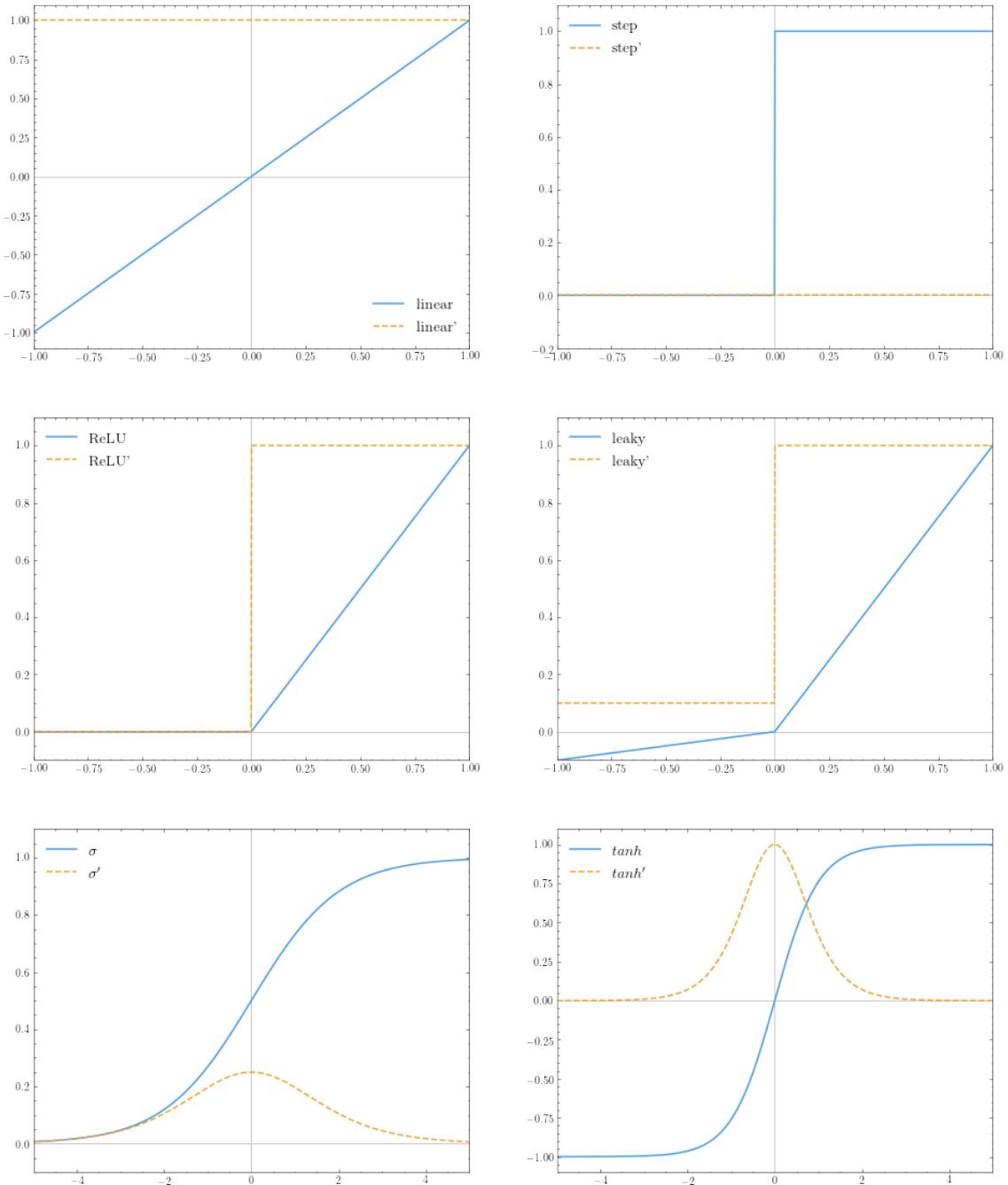


Figura 10: Gráficas de las funciones de activación

Cada neurona podría tener una función de activación asociada, pero por convención se usa un único tipo de función de activación por cada capa. Ambas soluciones aportan resultados similares y ninguna aporta en principio mejoras significativas al resultado del modelo. La principal diferencia es la complejidad que se añade a la hora de desarrollar una red si se quiere usar una función de activación distinta por neurona. En conclusión, siempre se opta por el uso de una única función de activación en todas las neuronas de una capa por simplicidad de desarrollo.

2.4.4. Trabajando con matrices

Para facilitar la explicación a partir de este punto, se procede a explicar distintas partes de la red y su notación matemática con alguna simplificación.

Se parte de la siguiente definición de red neuronal:

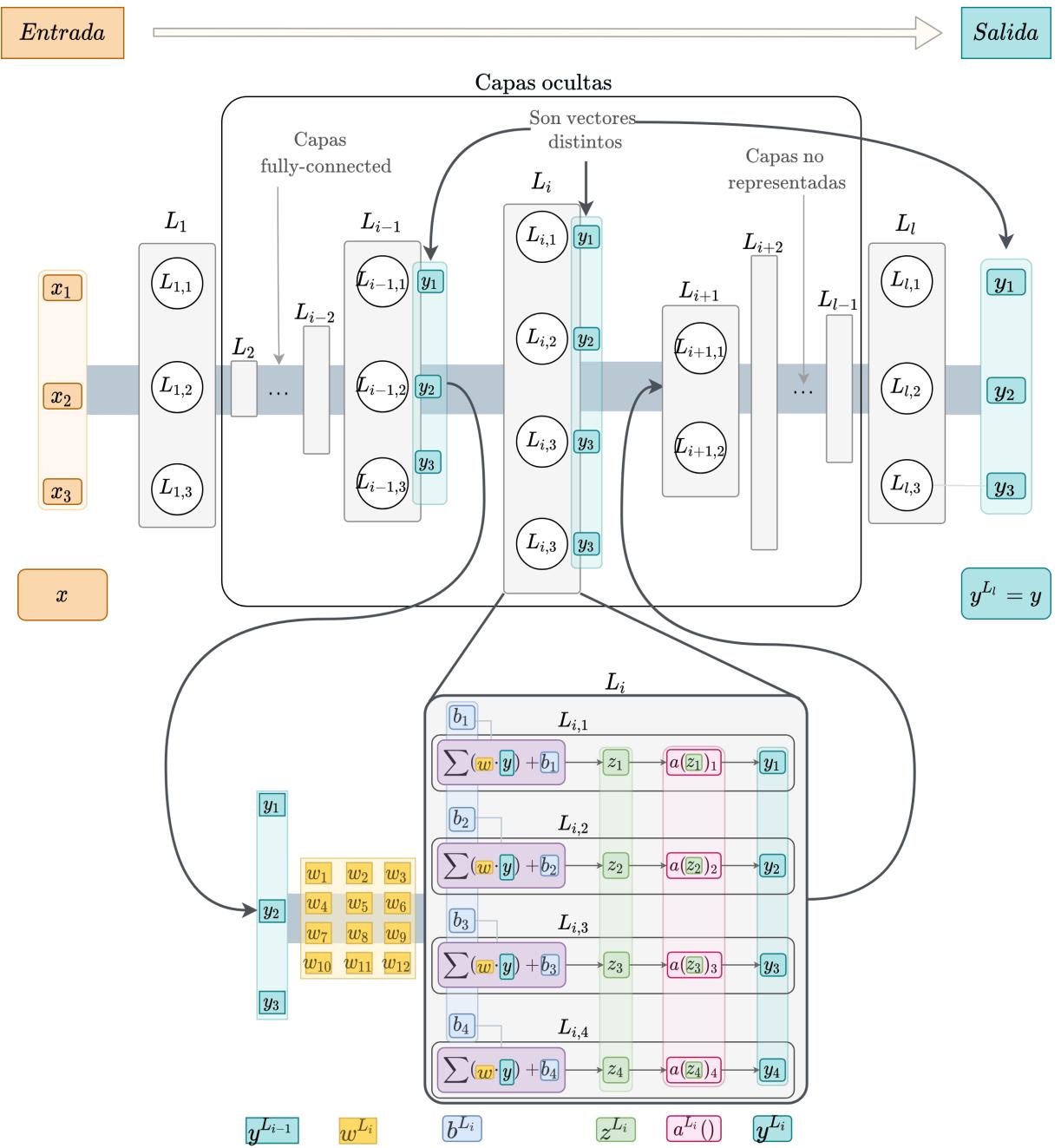


Figura 11: Red neuronal

La red representada necesita tres valores de entrada que devuelve un único valor, así el modelo será entrenado para que, dado un vector de tres elementos, el modelo otros tres elementos. El valor que devuelva el modelo será igual al valor y , calculado en la última

capa:

$$y = a^{L_3}(z_3) = w_3 \cdot a^{L_2}(z_2) + b_3$$

donde L_i = Última capa

w_j = Pesos en la última capa

b_k = Biases en la última capa

$a^{L_u}(z_u)$ = Resultado de la función de activación de la capa L_u sobre el vector z_u

$a^{L_2}(z_2)$ = Vector con los valores de la salida de penúltima capa

(21)

El número de elementos de z_{l-1} será el mismo número de neuronas que haya en la capa L_{l-1} . Esto ocurre $\forall z_i$. El vector z_{l-1} es calculado en la capa L_{l-1} de la siguiente forma:

$$z_{L_{l-1}} = w_{L_{l-1}} \cdot a^{L_{l-2}}(z_{L_{l-2}}) + b_{L_{l-1}} \quad (22)$$

Este proceso se repetirá hasta que z_1 sea calculado:

$$z_1 = w_1 \cdot x + b_1 \quad (23)$$

donde x = Vector de entrada

Para hacer referencia a una capa de la red se usará L_i , siendo i el índice de la capa ($i = 1$ para la primera capa, $i = 2$ para la segunda capa...). Para hacer referencia a la última capa, se usará L_l , la penúltima será L_{l-1} y así sucesivamente. Se puede añadir un segundo índice para hacer referencia a una neurona tal que $L_{i,j}$ y al igual que el índice de capa, si $j = 1$, hará referencia a la primera neurona de la capa, si $j = 2$, hará referencia a la segunda neurona de la capa y así respectivamente. A modo de ejemplo la tercera neurona de la segunda capa será referenciada como $L_{2,3}$ o $L_{l-1,3}$.

El vector w y valor b de $L_{2,3}$ serán referenciados tal que $w^{L_{2,3}}$ y $b^{L_{2,3}}$ respectivamente. Si se quiere hacer referencia a un valor del vector w en concreto se usará un tercer índice tal que $w_k^{L_{i,j}}$. Por ejemplo, $w_1^{L_{2,3}}$ hará referencia al primer peso de la tercera neurona de la segunda capa.

Sabiendo esta nueva notación, a partir de este punto se simplificarán las ecuaciones. Por ello, los parámetros de una neurona vendrán en un único vector añadiendo el valor b al vector ya existente w . Todos los vectores se agruparán formando una matriz W , donde cada fila será el vector asociado a una neurona. Es decir, los parámetros de una capa L_i

vendrá dado como una matriz a la que se hará referencia como W .

$$W^{L_i} = \begin{pmatrix} b^{L_{i,1}} & w_1^{L_{i,1}} & w_2^{L_{i,1}} & \dots & w_n^{L_{i,1}} \\ b^{L_{i,2}} & w_1^{L_{i,2}} & w_2^{L_{i,2}} & \dots & w_n^{L_{i,2}} \\ b^{L_{i,3}} & w_1^{L_{i,3}} & w_2^{L_{i,3}} & \dots & w_n^{L_{i,3}} \\ \vdots & \vdots & \vdots & & \vdots \\ b^{L_{i,m}} & w_1^{L_{i,m}} & w_2^{L_{i,m}} & \dots & w_n^{L_{i,m}} \end{pmatrix} \quad (24)$$

donde i = Índice de la capa en la red

n = Número de neuronas en L_{i-1} o longitud de vector de entrada si $i = 1$

m = Número de neuronas en la capa L_i

Como ejemplo, la matriz W^{L-1} de la red neuronal definida en la figura 11 sería la siguiente:

$$W^{L_{l-1}} = W^{L_2} = \begin{pmatrix} b_1 & w_1 & w_2 & w_3 \\ b_2 & w_4 & w_5 & w_6 \\ b_3 & w_7 & w_8 & w_9 \\ b_4 & w_{10} & w_{11} & w_{12} \end{pmatrix} \quad (25)$$

Usando el ejemplo de la figura 4, donde se usan valores específicos para w y b , las matrices de cada capa con los parámetros quedarían tal que:

$$W^{L_1} = \begin{pmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{pmatrix} \quad (26)$$

$$W^{L_2} = (-2 \ 3 \ -2) \quad (27)$$

Con esta nueva notación en una sola matriz se recoge los parámetros que necesita calcular la regresión. De hecho, esta matriz es los que el modelo tendrá que "aprender" para obtener mejores resultados. Esto se verá más en detalle en la sección 2.5.1.

En la ecuación 8 se calculaba el valor con la función $a()$ de la neurona. A partir de este punto, una función de activación recibirá un vector con los valores z de cada neurona y por tanto esta función $a()$ también devolverá un vector del mismo tamaño. En la primera capa de la red, este vector será calculado tal que:

$$y_1 = a^{L_1}(z) = a(W^{L_1}x) \quad (28)$$

Pero al definir la matriz W de esta forma, el producto $W^{L_1}x$ es imposible de calcular. Esto se puede demostrar usando el ejemplo de la figura 24. La dimensión de W^{L_1} es (3×6) . Sin embargo, el vector x con el que se va a multiplicar tiene una dimensión (1×5) . Para solucionar este problema, se añadirá un 1 al vector x , vector conocido como

x' y se realizará la transpuesta de dicho vector. De esta forma se podrá multiplicar la matriz W con dimensión 3×6 con un vector de dimensión 6×1 .

$$a_1^L(z) = a(W^{L_2}x')$$

donde $x' = [[1] \oplus x]^T = [1, x_1, x_2, \dots, x_n]^T$

$n = \text{Tamaño del vector } x$

$A \oplus B : \text{Concatenación de vectores } A \text{ y } B$

(29)

El resto de las capas no tiene este problema. El vector obtenido en la anterior capa ya cumple con las condiciones necesarias para poder realizar el productor de una matriz por un vector. El vector $a^{L_2}(z)$ usará la misma ecuación que 28, pero la variable x será el vector y calculado en la anterior capa:

$$a^{L_2}(z) = a(W^{L_2} \cdot a(W^{L_1}x')) \quad (30)$$

Se realizará este proceso hasta llegar a la última capa de la red. Se puede leer más información en la sección 2.4.5.

Actualizando también las ecuaciones 21, 22 y 23:

$$y = a^{L_3}(z_3) = W^{L_3} \cdot a^{L_2}(z_2) \quad (31)$$

$$z_2 = W^{L_2} \cdot a^{L_1}(z_1) \quad (32)$$

$$z_1 = W^{L_1} \cdot x' \quad (33)$$

Agrupando todas las ecuaciones en una sola, el valor de y en una red neuronal con tres capas viene dado por:

$$y = a(W^{L_3} \cdot (a(W^{L_2} \cdot (a(W^{L_1} \cdot x')^{L_1}))^{L_2}))^{L_3} \quad (34)$$

Esta ecuación es la que resuelve el modelo para poder obtener el valor y y es lo que se conoce como algoritmo de *forward-pass*.

2.4.5. Algoritmo de *forward-pass*

El algoritmo *forward-pass* es el algoritmo básico con el cual una red neuronal predice y . Como se ha visto en la ecuación 34, el algoritmo es al fin y al cabo una combinación de productos de matrices y funciones de activaciones. Se puede definir una ecuación general de la siguiente forma:

$$y = a(W^{L_l} \cdot (a(W^{L_{l-1}} \cdot \dots \cdot (a(W^{L_1} \cdot x')^{L_1})) \dots ^{L_{l-1}}))^{L_l} \quad (35)$$

Esta ecuación está computando el algoritmo *forward-pass*, que se resume en los siguientes pasos:

1. Se tendrá un vector x :
 - a) Si es la primera capa del modelo, x es el dado al modelo como argumento de entrada (vector x) pero traspuesto.
 - b) EOC x vendrá dado calculado por la capa anterior (vector y).
2. Se computa x' introduciendo un 1 en el inicio del vector.

$$x' = [[1] \oplus x]^T \quad (36)$$

3. Se calculará y :

$$y = a(Wx') \quad (37)$$

4. Si es la última capa, la predicción del modelo será y , e.o.c. volver al paso 1.b.

El mismo algoritmo en pseudocódigo:

Resultado: Cálculo vector y

$i = 0$;

n = Número de capas de la red;

x = Vector de entrada;

y = Vector de salida;

mientras $i \neq n$ **hacer**

si $i == 0$ **entonces**

| $y' = x^T$;

en otro caso

| $y' = y$;

fin

$x' = [[1] \oplus y']$;

$z = W^{L_i}x'$;

$y = a^{L_i}(z)$;

$i++$;

fin

Algoritmo 1: Algoritmo de *forward-pass*

Con esta ecuación es fácil ver que el uso de una función de activación por cada capa es necesaria. Como se ha explicado en la sección 2.4.3, sin el uso de las funciones de activación, un conjunto de capas puede ser compactada en una sola capa. A continuación, se muestra la demostración matemática, el *forward-pass* sin funciones de activación:

$$y = W^{L_l} \cdot W^{L_{l-1}} \cdot \dots \cdot W^1 \cdot x' = \left(\prod_{l;i=1}^{i=1} W^{L_i} \right) x' = W' x' \quad (38)$$

donde $W' = \left(\prod_{l;i=1}^{i=1} W^{L_i} \right)$

Esta red colapsa a una única capa cuyos parámetros vienen dados por la matriz de W' . Usando el ejemplo de la figura 11, si se decidiera no usar funciones de activación, la dimensión de la matriz W' sería de $(3, 1)$ que coincide con el tamaño del vector de entrada(x) y el tamaño del vector de salida(y).

$$\begin{aligned} \dim(W^1) &= (3, 6) \\ \dim(W^2) &= (6, 5) \\ \dim(W^3) &= (5, 1) \\ \dim(W^1 \cdot W^2) &= (3, 5) \\ \dim(W') &= \dim(W^1 \cdot W^2 \cdot W^3) = (3, 1) \end{aligned} \tag{39}$$

Recordando la siguiente regla para el producto de matrices para un producto de matrices $A \times B = C$:

$$\begin{aligned} \dim(A) &= (m, n) \\ \dim(B) &= (n, k) \\ \dim(C) &= (m, k) \end{aligned} \tag{40}$$

- La dimensión de la matriz C viene dada por el número de filas de A (m), y el número de columnas de B (n).
- El número de columnas de A (m) debe coincidir con el número de columnas de B (n), condición que se cumple puesto que se está trabajando con redes neuronales *fully-connected*, todas las neuronas de una capa están conectadas a las neuronas de la siguiente capa.

Volviendo a la ecuación 38 y realizando el mismo cálculo que se ha realizado en la ecuación 40 sobre la red que se quiera, se puede obtener el tamaño de y que será calculada por el modelo. El número de filas indicará el número de datos que se han predicho y el número de columnas es el número de etiquetas por cada dato.

En la red mostrada en la figura 11, tiene como entrada cinco variables. A modo de ejemplo, se podría usar esta red para crear el modelo del siguiente problema: Predecir el coste de una vivienda en una ciudad dado un conjunto de datos de vivienda en esa misma ciudad. Se quiere predecir el valor del precio a partir de las siguientes variables: número de habitaciones, tipo de vivienda, porcentaje de criminalidad, coordenadas geográficas y metros cuadrados. Esto se puede representar como una tabla donde cada fila es una vivienda.

	tipo	coords	gc¹	m2²	h³	precio
1	Estudio	(40.35717, -3.81053)	5	41	2	65.378
2	Apartamento	(40.53125, -3.72054)	2	49	3	79.519
3	Dúplex	(40.45990, -3.60053)	5	59	3	82.578
4	Apartamento	(40.43437, -3.74870)	1	84	3	99.701
5	Apartamento	(40.43646, -3.84071)	3	100	5	130.978
6	Chalé	(40.41922, -3.57909)	1	126	6	160.890
7	Apartamento	(40.47372, -3.58115)	4	94	4	215.000
8	Ático	(40.52186, -3.57051)	4	142	6	285.398
9	Chalé	(40.44979, -3.68003)	3	185	7	340.589
10	Dúplex	(40.49347, -3.67969)	2	209	8	426.000

¹ gc: Grado de criminalidad del barrio.

² m2: metros cuadrados.

³ h: Número de habitaciones.

Tabla 2: Ejemplo de *dataset* con viviendas

2.5. Redes neuronales artificiales. Entrenamiento

Las redes neuronales son algoritmos que se usan en el campo de la IA y tratan solventar problemas en los cuales se trabaja con gran cantidad de datos tratando de buscar patrones en ellos. Además, es una de las pocas alternativas a otros algoritmos que nos son capaces de tratar gran volumen de datos.

Los inicios de las redes neuronales se remontan a la década de 1950, cuando McCulloch y Walter Pitts[29] trabajaron en un modelo matemático que se asemejaba al comportamiento que conocían de una neurona. El científico Frank Rosenblatt, inspirado en este trabajo, desarrollo lo que se conoce como redes de perceptrones. Este fue el primer acercamiento a lo que hoy se conoce como redes neuronales[30].

2.5.1. Entrenamiento de una red

Como resumen de visto hasta ahora, para crear una red es necesario tener cuatro elementos distintos:

- Valores de entrada: Información con la que se va a predecir un dato o un conjunto de datos.
- Estructura de la red: Esta es una información que se debe de conocer a priori antes de crear el modelo. Tiene distintas propiedades:
 - Número de capas: A mayor número de capas, mayor será el tiempo que tardará el modelo en computar el vector de salida *y* porque mayor cantidad de cálculos tendrá que realizar. El número de capas junto con el número de neuronas por capa son parámetros importantes, puesto que un número muy bajo

en el modelo y este no tendrá una buena precisión. Por lo contrario un número muy alto puede producir lo que se conoce como *overfitting*(ver sección 2.5.7).

- Número de neuronas en cada capa: Cabe destacar que el número de neuronas en la última capa será el tamaño del vector y , es decir, el número de etiquetas que el modelo predecirá.
 - Función de activación por cada capa.
 - Arquitectura de la red.
- Las matrices W : Son matrices que recogen la información asociada a cada capa sobre los pesos w y *bias* b de cada neurona de la red.

Las matrices W son matrices con valores creados aleatoriamente. El proceso de entrenamiento de la red tratará de optimizar estas matrices para que el vector y resultante sea lo más preciso posible. Para poder entrenar un modelo es necesario tener previamente un *dataset* con un conjunto de vectores x y su valor real. La cantidad de datos que proporcionemos al modelo para que aprenda está relacionada con la precisión del modelo. Con el ejemplo descrito en la Tabla 2 se podrían usar las columnas *tipo*, *coords*, *gc*, *m2* y *h* para predecir el valor *precio*.

Básicamente la red inicializa la matriz W de forma aleatoria. El modelo dado un vector x realiza todos los cálculos de cada neurona y devuelve un valor y . y es lo que el modelo ha predicho. Este proceso es el que se conoce como *forward-pass*.

Para que la red aprenda, necesita primero saber si se ha equivocado y la magnitud del error. Si la magnitud de este error es muy grande, se deberá ajustar los valores de W para minimizar el error. Por lo contrario, si el error es pequeño, no se ajustarán mucho los valores de W porque realizando un ajuste en W puede provocar una ligera mejoría en dicha predicción, pero puede desajustar otras predicciones que el modelo ha hecho previamente y eran también bastantes precisas. La elegancia de este algoritmo es que encuentra un balance entre lo que es ajustar valores para que la red aprenda y al mismo tiempo no desajustar demasiado para que la red se descompense en su aprendizaje global. Es la aplicación de la imitación del aprendizaje humano, un error con una gran repercusión deberá modificar comportamientos futuros, un error sin repercusión podrá ser ignorado o incorporar cambios proporcionados a las repercusiones.

El error se cuantifica con una función llamada función de coste o función de pérdida explicada en la sección 2.5.2. A partir del valor del error, se usará un algoritmo que tratará de calcular la responsabilidad de cada neurona en dicho error y de esta forma poder ajustar los pesos y la *bias* asociadas a dicha neurona. Este algoritmo se llama *backpropagation* y es explicado en la sección 2.5.4.

Se puede pensar que este proceso se puede repetir infinitamente hasta tener un modelo perfecto, pero como se ve en la sección de 2.5.7 esto puede provocar que la red memorice el dataset que es usado para entrenar y no tenga la capacidad de generalizar. Por lo tanto,

no solo se trata de ejecutar el algoritmo, sino que hay que realizar ciertas optimizaciones y tener varios conceptos en cuenta como por ejemplo: la selección de función de activación, diseño del vector de entrada y salida, métricas a usar, entre otros.

Como analogía para entender el algoritmo de *backpropagation*, se puede usar la jerarquía de una empresa. Dicha empresa tras un trimestre desastroso elabora un resumen con los resultados. Estos resultados son equivalentes al error que el modelo produce. El jefe (la última capa de la red), tratará de rendir cuentas con los directivos. Estos directivos a su vez tratarán con otros directivos con menor responsabilidad y estos a su vez lo harán con jefes que estén por debajo suya y así sucesivamente hasta llegar al último nivel en la empresa (propagando el error hacia las capas más básicas). Posteriormente, la empresa elaborará un informe estudiando la responsabilidad de cada persona en el resultado del trimestre (que en el algoritmo de *backpropagation* es conocido como el vector gradiente). Ese informe llegará al departamento de recursos humanos y este, tratará de modificar el comportamiento de cada trabajador en la empresa en función de su responsabilidad en el error (algoritmo del descenso del gradiente).

En el siguiente diagrama se puede visualizar el proceso que se lleva a cabo para entrenar una red neuronal:

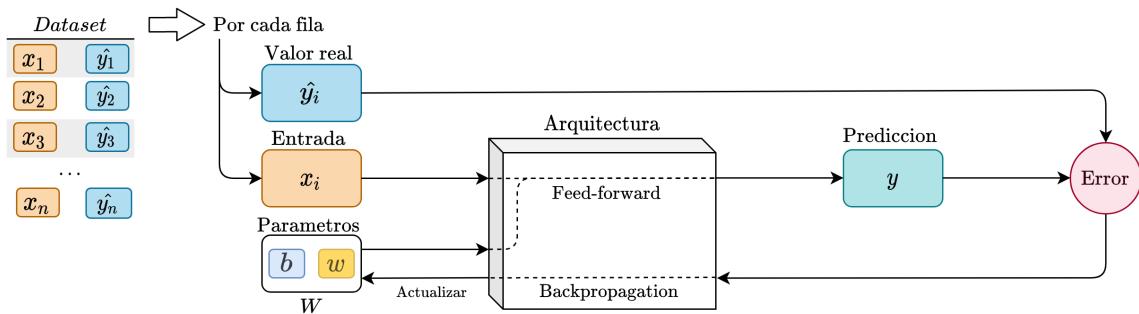


Figura 12: Proceso de entrenamiento de una red neuronal

El entrenamiento de una red es un proceso iterativo. En cada iteración, o también conocido como *epoch*, se tomarán un conjunto de datos del *dataset* de entrenamiento de forma aleatoria. Cada uno de estos conjuntos de datos tomados para cada epoch se le denomina *batch* y el tamaño del *batch* suele ser un parámetro puesto por el usuario. El tamaño de un *batch* suele ser de 32, 64 o 128 valores.

En cada *epoch*, el modelo solo trabajará con dichos datos. Predecirá un valor para cada una de las entradas y junto al valor real, la función de coste, el algoritmo de *backpropagation* y el descenso del gradiente, se irán ajustando de forma iterativa las matrices W .

2.5.2. Función de coste

Para que el modelo aprenda es necesario primero saber identificar errores en el proceso. La función de coste es una función que calculará el error que se está produciendo en un modelo. Esta función tendrá dos parámetros: El valor esperado y el valor calculado por el modelo. La diferencia entre ambos valores es lo que se conoce como error o pérdida, por eso esta función también es conocida como función de pérdida o función objetiva.

El error más simple es el error dado por la diferencia entre el valor esperado y el valor real:

$$c_i = \text{Valor}_{\text{real}} - \text{Valor}_{\text{esperado}} = \hat{y}_i - y_i,$$

donde c_i = El valor de la pérdida de la muestra

i = i-ésima muestra del dataset (41)

\hat{y}_i = Resultado del modelo

y_i = Valor real

Usando el ejemplo visto antes en la regresión de la Figura 6 el error para el dato i visto de forma gráfica sería el siguiente:

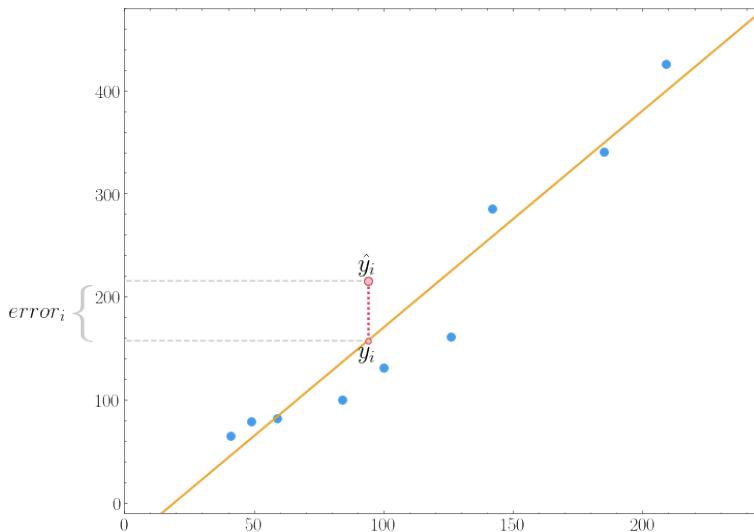


Figura 13: Error de una regresión para el dato i

En una red se necesita conocer el error de la capa y no el error de cada neurona. Para ello se aplicará alguna función que reciba un vector como entrada (los errores de cada neurona en una capa) y un único valor de salida (error de la capa). A continuación se listan algunas de las funciones más usadas[31]:

- Error Absoluto Medio (MAE) [32] : Es la métrica de error de regresión más simple de entender. Se toma el error absoluto de cada dato, para que los errores negativos y

positivos no se anulen. Luego se calculará la media aritmética. En realidad, el MAE describe la magnitud típica de los residuos. La ecuación es la siguiente:

$$\text{MAE} = c_i(y_1, \hat{y}_1) = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (42)$$

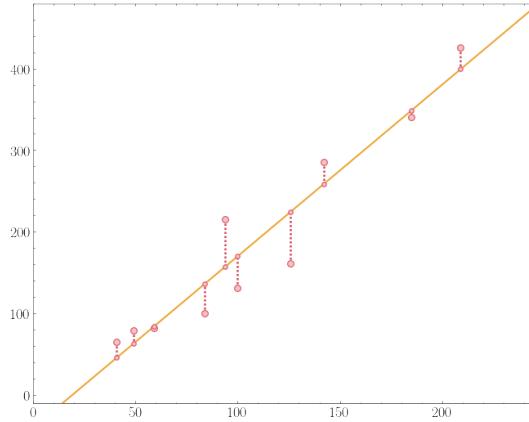


Figura 14: Visualización del MAE. El error es la media aritmética de todos los errores.

- Error cuadrático medio (MSE) [32]: Esta ecuación calcula la media de todos los errores elevados al cuadrado. Elevando al cuadrado se consigue penalizar con mayor intensidad a aquellos puntos que están más alejados de la estimación de la regresión lineal y con menor intensidad a los que se encuentren más cerca. Esta ecuación es muy usada cuando la tarea que se trata de resolver es de tipo regresión. La ecuación es la siguiente:

$$\begin{aligned} \text{MSE} = c_i(y_1, \hat{y}_1) &= \frac{1}{n} \sum_{i=1}^n (\sqrt{(\hat{y}_i - y_i)^2})^2 \\ &= \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \end{aligned} \quad (43)$$

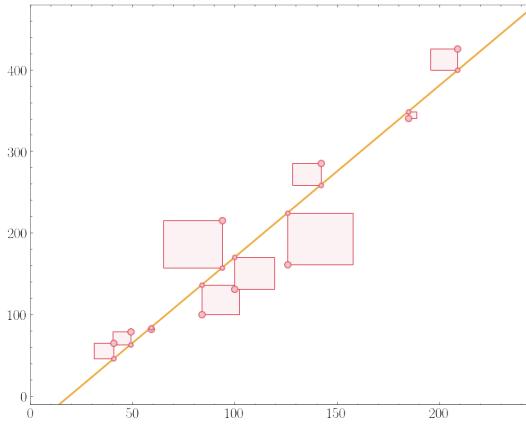


Figura 15: Visualización del MSE. El error es la media de las áreas de los cuadrados.

- Root Mean Squared Error (RMSE) [32] : Esta ecuación es la raíz cuadrada de MSE. Si se compara a nivel de valores, son intercambiables aunque utilizan distinta escala. La ecuación es la siguiente:

$$\text{Root Mean Squared Error (RMSE)} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2} \quad (44)$$

- Raíz del error medio cuadrático (MSLE) [32] : Esta ecuación es parecida a MSE. Se suele utilizar cuando se sabe previamente que los resultados están normalmente distribuidos y no se quiere que los errores grandes sean significativamente más penalizados que los pequeños. La ecuación es la siguiente:

$$\begin{aligned} \text{Raíz del error medio cuadrático (MSLE)} &= c_i(y_1, \hat{y}_1) = \frac{1}{n} \sum_{i=1}^n (\log y_i + 1 - \log \hat{y}_i + 1) \\ &= \frac{1}{n} \sum_{i=1}^n \left(\log \left(\frac{y_i + 1}{\hat{y}_i + 1} \right) \right)^2 \end{aligned} \quad (45)$$

- Pérdida de *Huber*[33]: Ya se sabe que el Error Medio Cuadrado (MSE) es mejor para aprender los valores atípicos en el conjunto de datos, por otro lado, el Error Medio Absoluto (MAE) es bueno para ignorar los valores atípicos. Pero en algunos casos, los datos que parecen atípicos no molestan y no deberían tener alta prioridad. La pérdida de Huber es una combinación MSE y MAE. Se usará δ para definir un sesgo para usar MAE o MSE. La ecuación es la siguiente:

$$\begin{aligned} \text{Huber_loss} &= c_i(y_1, \hat{y}_1) = \begin{cases} \text{MSE} & \text{si } |\hat{y}_i - y_i| \leq \delta, \\ \text{MAE} & \text{e.o.c.} \end{cases} \\ &= \begin{cases} \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 & \text{si } |\hat{y}_i - y_i| \leq \delta, \\ \delta \left(\frac{1}{n} \sum_{i=1}^n |\hat{y}_i - y_i| - \frac{\delta}{n} \right) & \text{e.o.c.} \end{cases} \end{aligned} \quad (46)$$

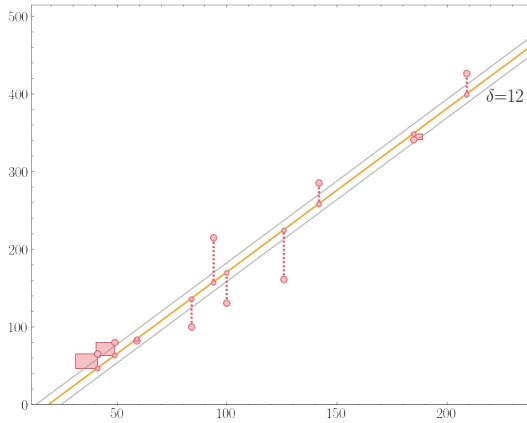


Figura 16: Visualización de la pérdida de Huber. Se calcula MSE si $|\hat{y} - y| \leq \delta$, en otro caso se calcula MSE.

- Porcentaje medio de error absoluto (MAPE) [32] : Es el porcentaje equivalente de MAE. La ecuación es igual a la de MAE, pero con ajustes para convertir los valores en porcentajes. Permiten ver la distancia entre los resultados del modelo y el resultado real mostrando el dato de manera más fácil de interpretar para los seres humanos. La ecuación es la siguiente:

$$\text{Porcentaje medio de error absoluto (MAPE)} = c_i(y_1, \hat{y}_1) = \frac{100\%}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right| \quad (47)$$

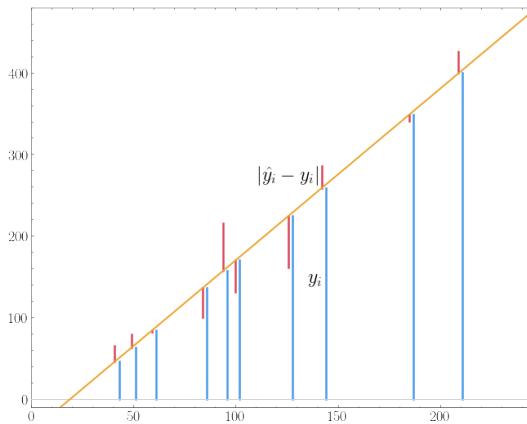


Figura 17: Visualización del MAPE. El error es la media de las proporciones de los errores respecto al valor y .

- Porcentaje medio de error (MPE) [32] : Es exactamente que MAPE, pero sin el valor absoluto. Dado que los errores positivos y negativos se anulan, no se puede hacer ninguna declaración sobre el rendimiento general de las predicciones del modelo.

Sin embargo, si hay más errores negativos o positivos, este sesgo se mostrará en el MPE. A diferencia del MAPE y del MAE, el MPE es útil porque permite ver si el modelo sistemáticamente subestima (más errores negativos) o sobreestima (errores positivos). La ecuación es la siguiente:

$$\text{MPE} = c_i(y_1, \hat{y}_1) = \frac{100\%}{n} \sum_{i=1}^n \frac{y_i - \hat{y}_i}{y} \quad (48)$$

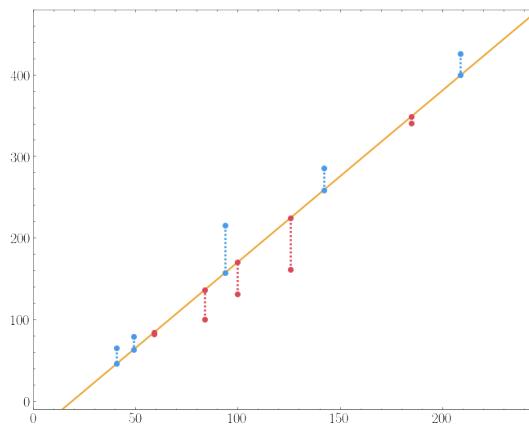


Figura 18: Visualización del MPE. Muestra la cantidad de errores que son positivos y la cantidad que son negativos.

- Pérdida de entropía cruzada: Se utiliza explícitamente para comparar una probabilidad de "verdad fundamental" (y u "objetivos") y alguna distribución predicha (\hat{y} o "predicciones"). Tiene como objetivo calcular la media de las probabilidades de que un valor pertenece a una clase o a otra, por lo que es muy útil para problemas de clasificación. Es muy usada cuando se usa la función de activación *softmax*, puesto que esta, también trabaja con probabilidades.

$$c_i(y_1, \hat{y}_1) = - \sum_i y_{i,j} \log(\hat{y}_{i,j}) \quad (49)$$

donde j = Índice de la probabilidad "verdadera"

La probabilidad "verdadera" es un vector con todos los valores a 0 excepto uno de ellos que tiene el valor igual a 1. Este tipo de vector es conocido como vector *one-hot*, donde un valor es *hot* si es igual a 1 o *cold* si es igual a 0. Cuando se comparan los resultados del modelo con un vector *one-hot* usando la entropía cruzada, los valores igual a 0 no se usan, y la pérdida de logaritmo de la probabilidad del objetivo se multiplica por 1, haciendo que el cálculo de la entropía cruzada sea relativamente simple. Este es también un caso especial del cálculo de la entropía cruzada, llamado entropía cruzada por categorías.

Un ejemplo de un vector *one-hot* sería el siguiente: [0, 1, 0, 0] donde por ejemplo se representarían las siguientes clases: Perro, gato, caballo y elefante. El 1 representa que el dato dado al modelo representa a un gato. Este tipo de vectores se usan mucho en tareas de clasificación y en Procesado del Lenguaje Natural (NLP).

Hay un subtipo denominado pérdida de entropía cruzada binaria. Este tipo de error se puede calcular cuando se intenta clasificar solo con dos tipos: 0 o 1. Normalmente se usa como si fuese un valor booleano en un lenguaje de programación. Un `true` si es del tipo A o un `false` si no es del tipo A. Por ejemplo: gato o no gato o en interior o en exterior. La ecuación matemática es la siguiente:

$$\begin{aligned} c_{i,j}(y_1, \hat{y}_1) &= (y_{i,j})(-\log(\hat{y}_{i,j})) + (1 - y_{i,j})(-\log(1 - \hat{y}_{i,j})) \\ &= -y_{i,j} \cdot \log(\hat{y}_{i,j}) - (1 - y_{i,j}) \cdot \log(1 - \hat{y}_{i,j}) \end{aligned} \quad (50)$$

2.5.3. Minimizando el error

Una vez elegida alguna de las funciones de coste, el objetivo es minimizar el error y así mejorar el modelo:

$$\begin{aligned} W^* &= \underset{W}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n c(a(x^{(i)}; W), y^{(i)}) \\ &= \underset{W}{\operatorname{argmin}} c_i(W) \end{aligned} \quad (51)$$

donde $x^{(i)}$ = Vector de entrada para el dato i

$y^{(i)}$ = Vector real para el dato i

W = Matrices con los pesos y bias de cada cada

Para calcular el mínimo de una función, se necesita calcular la derivada de dicha función e igualarla a 0. Minimizando la función de coste, es decir, derivando la función de coste, también se minimiza el error. Para ello hay distintas formas de minimizar el error:

- Minimizar función de coste

Este fue el algoritmo usado en las redes de perceptrones para calcular la matriz W . El funcionamiento de este algoritmo es sencillo. Partiendo de la definición de una función de coste cualesquiera que permita cuantificar el error del modelo, se tratará de minimizar dicho valor. Una de las ecuaciones más populares es la que se conoce como el Mínimo Cuadrado Ordinarios. Esta función parte de la derivada de MSE:

$$\begin{aligned} L_i &= (\hat{y}_i - y_i)^2 \\ &= (y - x \cdot W)' \cdot (y - x \cdot W) \\ &= y'y - W'x'y - y'xW + W'x'xW \end{aligned} \quad (52)$$

Derivando y despejando:

$$\begin{aligned} L'_i() &= -2x'y + 2x'xW \\ W^* &= (x'x)^{-1}x'y \end{aligned} \quad (53)$$

Esta ecuación permite el cálculo de la matriz W óptima posible solo usando los valores de entrada y de salida de modelo como argumentos. Aunque este algoritmo de entrenamiento tiene varias limitaciones:

- No es extensible a redes más complejas.
- El cálculo de la matriz inversa es muy costoso computacionalmente.
- Se ha usado el error mínimo cuadrado, una de las más simple, puesto que es una función con forma convexa y por tanto una derivada fácil de calcular, pero hay otras funciones de coste que no permiten minimizar el error con esta técnica.

Estas limitaciones, demostradas matemáticamente en el libro "*Perceptron*"[34] de Minsky y Papert (1969), provocó un corte repentino en la financiación de proyectos de inteligencia artificial y más específicamente en aquellos relacionados con los sistemas de redes neuronales durante un periodo de más de 15 años conocidos como el invierno de la inteligencia artificial.

■ Descenso del gradiente

Este algoritmo no es una fórmula como los mínimos cuadrados ordinarios, sino un método iterativo que poco a poco va minimizando el error. De algún modo se puede asimilar a como aprendemos los seres humanos: No con una única fórmula, sino a través de la experiencia reduciendo nuestros errores con el tiempo.

El cálculo del gradiente es un proceso que se puede representar de la siguiente forma:

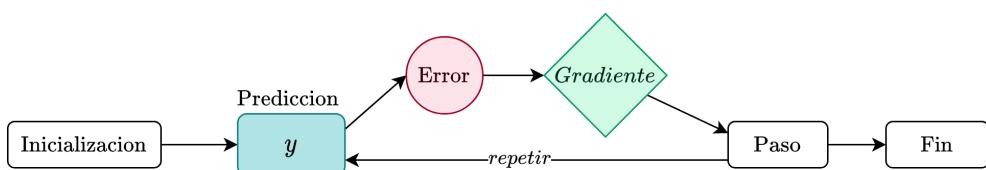


Figura 19: Proceso iterativo para minimizar el error

El método de los OLS minimiza el error igualando la derivada a 0 para poder hallar los mínimos de MSE. Con esto, se pueden calcular los mínimos locales y globales, pero también se puede calcular máximo locales, puntos de inflexión o puntos de silla provocando un sistema de ecuaciones grande y bastante ineficiente de resolver [34].

Una forma de entender este método es el siguiente: Una persona se encuentra en un terreno montañoso y el objetivo de esta persona es llegar al punto más bajo. Para ello, analizará el terreno donde se encuentra y evaluará la pendiente y se moverá hacia donde la pendiente desciende con mayor intensidad. Descenderá una cantidad de pasos y repetirá el proceso: analizar la inclinación y descender. Esto se repetirá hasta que llegue a lo más abajo posible y no haya forma alguna de seguir bajando. Esta es la lógica del algoritmo del descenso del gradiente.

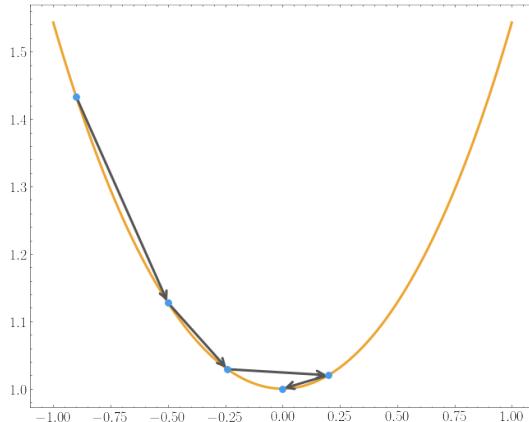


Figura 20: Ejemplo gráfico de la iteración del descenso del gradiente

El terreno en esta metáfora sería la función de coste. La persona es el valor calculado por la función de coste, el cual se quiere minimizar y por ello se evalúa la pendiente o gradiente solo en ese punto y de ese modo no se depende de la derivada de la función de coste sino de la derivada parcial respecto a cada uno de los valores de entrada en la neurona. El número de pasos que la persona bajará es un valor conocido como tasa de aprendizaje (*learning rate* en inglés) y es un parámetro más de la red neuronal.

En la figura 22, hay dos ejes, representando una neurona que tiene solo un argumento de entrada. El eje de ordenadas representa el error de la neurona. Se ha representado un espacio bidimensional, pero se puede usar cualquier dimensión puesto que el número de entradas de la neurona no está limitado. Usando un espacio tridimensional, la función de coste sería una superficie irregular. De hecho, el vector ∇f siempre tendrá como mínimo dos valores, un peso w y la bias b .

En este trabajo no se explica cómo calcular la derivada de una función o derivada parcial(σ). Dicho esto, matemáticamente, el proceso de este algoritmo es el siguiente:

1. Se inicializa los pesos w y bias b de la neurona quieren ajustar de forma aleatoria:

$$N(0, \sigma_2) \quad (54)$$

2. Se realiza un bucle hasta la convergencia:

- a) Se calculan derivadas parciales para cada uno de los parámetros. Una derivada parcial mide cuánto impacto tiene una única variable de entrada en la salida de la función y se calcula igual que una derivada, lo único que se repite la derivada para cada variable de entrada. Cada uno de esos valores indicará cuál es la pendiente en el eje de dicho parámetro relacionado a un único valor de entrada.

$$\begin{aligned}\nabla f &= \nabla f_b \oplus \nabla f_w \\ \nabla f_b &= \left(\frac{\partial c}{\partial b} \right) \\ \nabla f_w &= \left(\frac{\partial c}{\partial w_1}, \quad \frac{\partial c}{\partial w_2}, \quad \dots, \quad \frac{\partial c}{\partial w_n} \right)\end{aligned}\tag{55}$$

Conjuntamente todas las direcciones, es decir, todas las derivadas parciales conforman un vector que indica la dirección a la que la pendiente asciende, este vector es también conocido como gradiente(∇f) y el cual tendrá el mismo número de elementos que el vector de entrada y cada valor contendrá la solución a la derivada parcial con respecto a cada uno de los valores de entrada.

El objetivo de esta función es minimizar y no maximizar, por lo que se negará el gradiente para indicar la dirección en la que la pendiente desciende.

$$\nabla f = -\nabla f_b \oplus \nabla f_w\tag{56}$$

- b) Se actualizan los pesos con los nuevos valores del gradiente negativo multiplicado por la tasa de aprendizaje. La tasa de aprendizaje es simplemente un valor que determina cuán grande será el paso en cada iteración que se verá con mayor profundidad en la sección 2.5.5:

$$\begin{aligned}b^* &= b - \eta * \nabla f_b \\ w^* &= w - \eta * \nabla f_w\end{aligned}\tag{57}$$

Al igual que ocurría con el algoritmo de *feed-forward*, el descenso del gradiente trabajará por capas, por lo que este último paso se debería representar de la siguiente forma:

$$W^* = W - \eta * \nabla f\tag{58}$$

Esta ecuación representa que se actualizarán los pesos de todas las neuronas en una capa.

Este método se aplicará a todas las neuronas de la red. Dado un error, el descenso del gradiente obtendrá un vector que contendrá la derivada de los parámetros respecto al coste. Es decir, el gradiente será un vector con distintos valores, cuantos más grandes

sea dicho valor, más corrección se debe aplicar al parámetro que corresponde. Visto gráficamente se puede ver de la siguiente forma:

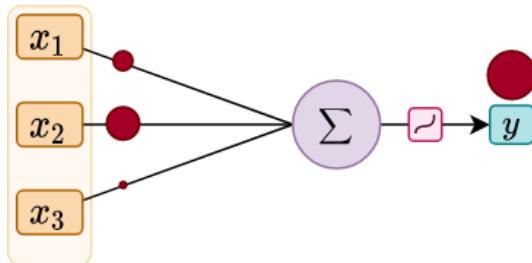


Figura 21: Representación del descenso del gradiente en una neurona. Lo rojo es la representación del error.

Las derivadas parciales que se deben resolver para obtener el vector gradiente ∇f , indica como varía el coste ante el cambio de los parámetro w y b . A continuación, se muestra una imagen de las distintas partes de las derivadas parciales:

¿Cómo varía el coste ante un cambio de los parámetros w y b?

$$\frac{\partial \textcolor{red}{c}_i}{\partial w} \quad \frac{\partial \textcolor{red}{c}_i}{\partial b}$$

Figura 22: Derivadas parciales usadas en el descenso del gradiente.

2.5.4. Backpropagation

En 1986, Rumelhart junto con otros investigadores publicaron “*Learning representation by back-propagating errors*” [35], un trabajo que volvió dar popularidad a las redes neuronales. El trabajo, basado en otros trabajos basados en diferenciación automáticas mostró experimentalmente como usando un nuevo algoritmo de aprendizaje se podría conseguir que una red neuronal se auto ajustará sus parámetros para así aprender una representación interna de la información que estaba procesando, un algoritmo conocido como *backpropagation*.

Este algoritmo es adaptable a cualquier modelo y con él, dio por finalizado lo que históricamente se conoce como Invierno de la IA consiguiendo nuevas financiaciones y nuevos proyectos en el campo del Aprendizaje profundo (DL).

La predicción de un modelo se obtiene usando un algoritmo llamado *forward-pass* como se ha explicado en la Sección 2.4.5. Dicho valor debe ajustarse lo mejor posible

al valor real. Para ello, se debe ir ajustando en un método iterativo las matrices W en un proceso que se conoce como entrenamiento (ver Sección 2.5.1). En cada iteración del entrenamiento se irá ajustando las matrices W haciendo uso primero de la función de coste, que mostrará la magnitud del error (ver Sección 2.5.2), posteriormente ese error se irá propagando hacia atrás en el modelo para conocer la responsabilidad de cada neurona usando un algoritmo conocido como *backpropagation* y finalmente, por cada neurona se calculará el vector gradiente ∇f que representa como ha afectado la neurona al resultado final y con ello se ajustarán sus distintos pesos w y bias b (ver Sección 2.5.3).

El algoritmo de *backpropagation* se puede representar gráficamente:

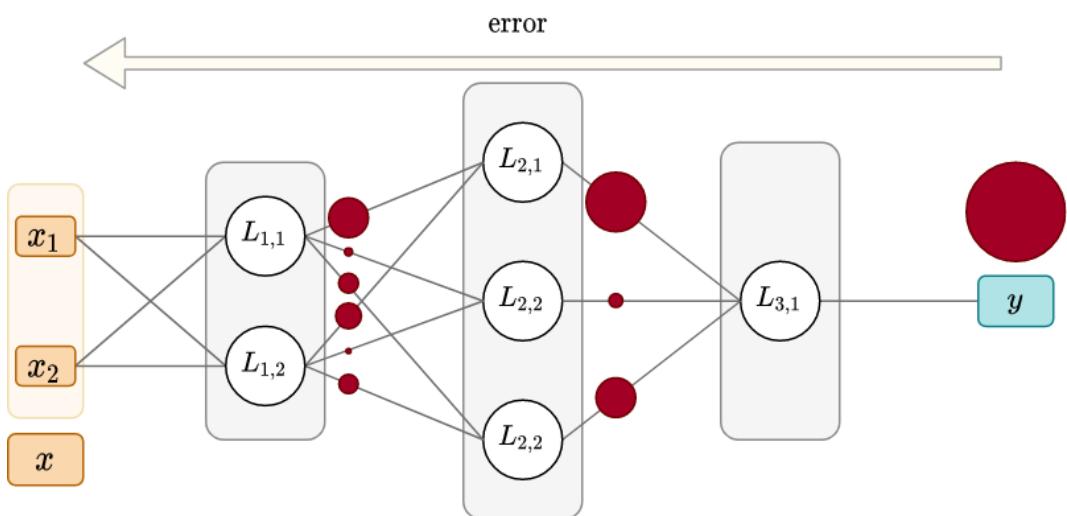


Figura 23: Representación del algoritmo de backpropagation.

En esta red se representa con círculos rojos el error de cada neurona. El error calculado por la función de coste es el error más grande que es el que está relacionado con la salida del modelo. A partir de ese punto, se retropropagará hacia atrás y se computará que parte del error pertenece a cada neurona hasta la primera capa. Por ejemplo, en la capa L_2 , la principal neurona que tiene parámetros desajustados es $L_{2,1}$. A su vez, ese error principalmente tiene parte de la culpa debido a la neurona $L_{1,1}$. Por lo tanto, es lógico pensar que habría que ajustar neuronas como $L_{2,1}$ o $L_{1,1}$, sin embargo, dejar tal como están neuronas como $L_{1,2}$ o $L_{2,2}$.

A continuación, se define la red neuronal más simple que se puede construir con una sola neurona oculta y una neurona de salida.

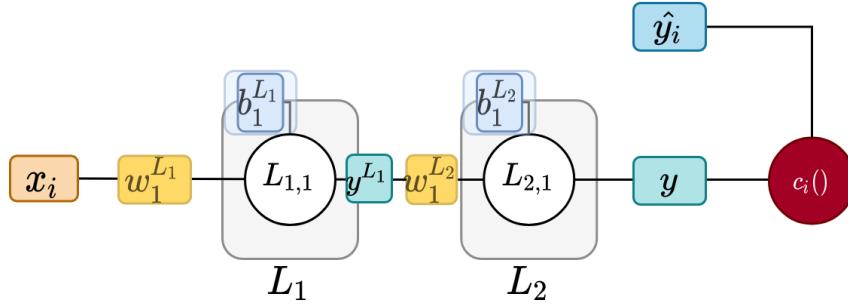


Figura 24: Red neuronal básica.

Para realizar el algoritmo de *backpropagation*, primero se debe de computar la pérdida con la función c_i . Posteriormente, el algoritmo de *backpropagation* "distribuirá" la pérdida del modelo a las capas anteriores (siempre se empieza por la última capa). Esta computación expresa cómo de importante son las matrices W indicando cuánto impacto tiene en el modelo. Por ejemplo, si se cambia algún valor de W_2 de alguna forma, se podrá saber cómo afecta al resultado de la predicción del modelo. Matemáticamente:

$$\begin{aligned}\frac{\partial c}{\partial w^{L_2}} &= \frac{\partial c(a(z^{L_2}))}{\partial w^{L_2}} \\ \frac{\partial c}{\partial b^{L_2}} &= \frac{\partial c(a(z^{L_2}))}{\partial b^{L_2}}\end{aligned}\tag{59}$$

El numerador es una composición de funciones, y es básicamente el cálculo del algoritmo *forward-pass* (ver sección 2.4.5). Para calcular la derivada parcial de una composición de funciones, hay que hacer uso de una herramienta de cálculo conocida como la regla de la cadena (*chain rule*).

Un ejemplo para entender la *chain rule* sería el siguiente. Se sabe que el Sol es 100 veces más grande que la Tierra. A su vez, la Tierra es 4 veces más grande que la Luna. Sabiendo esas dos relaciones, se puede saber cuánto es el Sol más grande que la Luna. Para ello simplemente multiplicamos $100 \times 4 = 400$. El sol por lo tanto es 400 veces más grande que la Luna. Definiendo este problema con ecuaciones:

$$\begin{aligned}\frac{\partial T_{\text{Sol}}}{\partial T_{\text{Tierra}}} &= 100 & \frac{\partial T_{\text{Tierra}}}{\partial T_{\text{Luna}}} &= 4 \\ \frac{\partial T_{\text{Sol}}}{\partial T_{\text{Luna}}} &= \frac{\partial T_{\text{Sol}}}{\partial T_{\text{Tierra}}} \cdot \frac{\partial T_{\text{Tierra}}}{\partial T_{\text{Luna}}} & &= 400\end{aligned}$$

Se le llama la *chain rule* porque se encadena en relación con el dato en común, en este caso T_{Tierra} es la parte común que en un lado se encuentra en el denominador y en otro en el numerador. Por lo tanto, para derivar la composición de funciones vista en la Ecuación 68, simplemente hay que multiplicar cada una de las derivadas intermedias. Recordando

las ecuaciones del algoritmo de *forward-pass* y la composición de funciones para calcular el error del modelo para la red mostrada en la Figura 28:

$$z^{L_2} = W^{L_2} \cdot y^{L_1} + b^{L_2} \quad c(a^{L_2}(z^{L_2})) \quad (60)$$

Se calcula como se propaga el error de la capa L_2 a la capa L_1 :

$$\begin{aligned} \frac{\partial c}{\partial w^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial w^{L_2}} \\ \frac{\partial c}{\partial b^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial b^{L_2}} \end{aligned} \quad (61)$$

Estas derivadas son fáciles de calcular y la explicación de cada derivada sería la siguiente:

- 1º derivada. Derivada de la función de activación respecto al coste:

$$\frac{\partial c}{\partial a^{L_2}} \quad (62)$$

Como varia el coste de la red (es la última capa) cuando se varía la salida de la función de activación de la red. En otras palabras, se calcula la derivada de la función de coste con respecto a la salida de la red neuronal. Es necesario, por tanto, saber la derivada de la función de coste usada.

- 2º derivada. Derivada de la activación respecto a z :

$$\frac{\partial a^{L_2}}{\partial z^{L_2}} \quad (63)$$

Trata de reflejar como varía la salida de la neurona cuando se varía la suma ponderada de la neurona. La derivada que hay que usar es la derivada de la función de activación que se pueden ver en la sección 2.4.3. Derivada de la suma ponderada respecto a los valores de los pesos y bias.

- 3º derivada: Derivada de z respecto a los parámetros:

$$\frac{\partial z^{L_2}}{\partial w^{L_2}} \quad \frac{\partial z^{L_2}}{\partial b^{L_2}}$$

Indica como varía la suma ponderada z con respecto a una variación de los parámetros. El parámetro b es un valor independiente por lo que su derivada es una constante igual a 1. Por otro lado, la derivada $\frac{\partial z^{L_2}}{\partial w^{L_2}}$ depende de la anterior capa.

$$\frac{\partial z^{L_2}}{\partial w^{L_2}} = a^{L_1} \quad \frac{\partial z^{L_2}}{\partial b^{L_2}} = 1$$

Partiendo de esta definición, se puede unir la 1º y 2º derivada en una sola:

$$\frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} = \frac{\partial c}{\partial z^{L_2}} \quad (64)$$

Esta derivada indica en qué grado se modifica el error cuando se produce un cambio en el error cuando se produce un pequeño cambio en la suma de las neuronas de la capa. Si el valor es grande es que ante un pequeño cambio en cualquiera de las neuronas de las capas en el valor z , se verá reflejado en el resultado del modelo. Por el contrario, si el valor es pequeño, un gran cambio en cualquiera de las neuronas en el valor de z de no afectará al resultado final. En resumen, esta derivada muestra cómo afecta la capa (o alguna neurona en especial puesto que es un vector) en el resultado final y por lo tanto al error de la red y de esa forma el algoritmo del descenso del gradiente modificará los parámetros para poder obtener el mejor resultado posible. A esta derivada también se le conoce como el error imputado a la capa y se representa con δ :

$$\frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} = \frac{\partial c}{\partial z^{L_2}} = \delta^{L_2} \quad (65)$$

Con esta nueva definición, se puede reescribir la Ecuación 61 de la siguiente forma:

$$\begin{aligned} \frac{\partial c}{\partial w^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial w^{L_2}} \\ &= \delta^{L_2} \cdot \frac{\partial z^{L_2}}{\partial w^{L_2}} = \delta^{L_2} \cdot a^{L_1} \end{aligned} \quad (66)$$

$$\begin{aligned} \frac{\partial c}{\partial b^{L_2}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial b^{L_2}} \\ &= \delta^{L_2} \cdot \frac{\partial z^{L_2}}{\partial b^{L_2}} = \delta^{L_2} \cdot 1 \end{aligned} \quad (67)$$

Pero el modelo no solo depende de la segunda capa y de W_2 , también depende de la primera capa y de W_1 . Es aquí donde mana la belleza del algoritmo de *backpropagation*. El error se puede retropropagar de manera similar usando el mismo razonamiento. Primero se muestran la composición de funciones:

$$\begin{aligned} \frac{\partial c}{\partial w^{L_1}} &= \frac{\partial c(a(z^{L_2}(a(z^{L_1}))))}{\partial w^{L_2}} \\ \frac{\partial c}{\partial b^{L_1}} &= \frac{\partial c(a(z^{L_2}(a(z^{L_1}))))}{\partial b^{L_2}} \end{aligned} \quad (68)$$

Usando la *chain rule* se divide en distintas partes:

$$\begin{aligned} \frac{\partial c}{\partial w^{L_1}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \cdot \frac{\partial a^{L_1}}{\partial z^{L_1}} \cdot \frac{\partial z^{L_1}}{\partial x} \\ \frac{\partial c}{\partial b^{L_1}} &= \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \cdot \frac{\partial a^{L_1}}{\partial z^{L_1}} \cdot \frac{\partial z^{L_1}}{\partial 1} \end{aligned} \quad (69)$$

En realidad, de las 6 derivadas mostradas, sólo haría falta calcular una de ellas. Las derivadas $\frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}}$ ya han sido calculadas y son igual a δ^{L_2} . En cuanto a la derivada $\frac{\partial z^{L_1}}{\partial x}$, simplemente es obtener el resultado de la anterior capa o bien usar x si es la primera capa como es este caso. La derivada $\frac{\partial z^{L_1}}{\partial 1}$ es un valor constante por lo que no pasa nada. Finalmente, la derivada $\frac{\partial a^{L_1}}{\partial z^{L_1}}$ es simplemente la derivada de la función de activación de dicha capa. Es por esta razón por lo que en la sección 2.4.3 se explicaba que es mejor usar una única función de activación para toda la capa y así poder facilitar los cálculos.

La única derivada que queda por explicar es $\frac{\partial z^{L_2}}{\partial a^{L_1}}$ la cual expresa como varía la suma ponderada de una capa cuando se varía el vector de entrada de dicha capa. El cálculo de esta derivada es simplemente la matriz de parámetros W^{L_1} que conecta ambas capas. Básicamente, esta derivada mueve el error de la capa L_2 a la capa L_1 distribuyendo el error en función de las ponderaciones de las conexiones. Al igual que se hizo con la anterior capa, con esta capa también se puede definir el error imputado:

$$\frac{\partial c}{\partial a^{L_1}} = \frac{\partial c}{\partial a^{L_2}} \cdot \frac{\partial a^{L_2}}{\partial z^{L_2}} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \cdot \frac{\partial a^{L_1}}{\partial z^{L_1}} = \delta^{L_1} \quad (70)$$

Por lo tanto, la Ecuación 69 quedaría de la siguiente forma:

$$\begin{aligned} \frac{\partial c}{\partial w^{L_1}} &= \delta^{L_1} \cdot \frac{\partial z^{L_1}}{\partial x} \\ \frac{\partial c}{\partial b^{L_1}} &= \delta^{L_1} \frac{\partial z^{L_1}}{\partial 1} \end{aligned} \quad (71)$$

En resumen, se ha podido explicar cómo propagar el error a cualquier capa de la red y calcular la derivada respecto a los parámetros con las siguientes ecuaciones:

$$\begin{aligned} \frac{\partial c}{\partial w^{L_1}} &= \delta^{L_1} \cdot \frac{\partial z^{L_1}}{\partial x} \\ \frac{\partial c}{\partial b^{L_1}} &= \delta^{L_1} \\ \frac{\partial c}{\partial w^{L_2}} &= \delta^{L_2} \cdot \frac{\partial z^{L_2}}{\partial a^{L_1}} \\ \frac{\partial c}{\partial b^{L_2}} &= \delta^{L_2} \end{aligned} \quad (72)$$

Visto el algoritmo de *backpropagation* para la red definida en la Figura 28, el algoritmo se divide en diferentes pasos:

1. Inicializar índice para saber en la capa que se va a computar

$$i = l; l > 0 \quad (73)$$

2. Cálculo del error imputado de la capa:

a) Si $i = l$:

$$\delta^{L_i} = \frac{\partial c}{\partial a^{L_i}} \cdot \frac{\partial a^{L_i}}{\partial z^{L_i}} \quad (74)$$

b) e.o.c:

$$\delta^{L_i} = \delta^{L_{i+1}} \cdot \frac{z^{L_{i+1}}}{\partial a^{L_i}} \cdot \frac{\partial a^{L_i}}{\partial z^{L_i}} \quad (75)$$

3. Cálculo de las derivadas sobre los parámetros b y W :

a) Cálculo sobre b :

$$\frac{\partial c}{\partial b^{L_i}} = \delta^{L_i} \quad (76)$$

b) Cálculo sobre w :

1) Si $i = 1$:

$$\frac{\partial c}{\partial w^{L_i}} = \delta^{L_i} \cdot x \quad (77)$$

2) e.o.c:

$$\frac{\partial c}{\partial w^{L_i}} = \delta^{L_i} \cdot a^{L_{i-1}} \quad (78)$$

4. Actualizar el índice:

$$i = i - 1 \quad (79)$$

5. Si $i > 0$ volver al paso 2

Con esto se ha conseguido obtener las derivadas parciales de cada capa de la red con lo que se podrá calcular las nuevas matrices W usando el método del descenso del gradiente explicado en la Sección 2.5.3.

2.5.5. Tasa de aprendizaje

Seleccionar una tasa de aprendizaje adecuada para el modelo es un paso fundamental a la hora de diseñar una red neuronal y una mínima modificación de este valor puede tener un gran impacto en el modelo final.

Si se selecciona una tasa de aprendizaje muy pequeña, significa que no se fía del resultado del gradiente y por lo tanto en cada iteración el cambio que sufrirá la matriz W será pequeño y el algoritmo se podrá atascar en alguno de los puntos locales mínimos porque el cambio entre la W antigua y la nueva W no está siendo tan drástico como debería para no acabar en estos mínimos locales. Por el contrario, si se selecciona una tasa de aprendizaje muy grande, el algoritmo se sobrepasará por completo y divergirá.

Por lo tanto, el valor para la tasa de aprendizaje no debe de ser ni muy pequeño para que el algoritmo no se atasque ni muy grande para que el modelo pueda convergir. Una

forma de elegir una buena tasa de aprendizaje es probar varios valores y estudiar qué valor funciona mejor. Este algoritmo es conocido como Descenso de gradiente estocástico (SGD) [36], pero otra opción es usar algún optimizador.

2.5.6. Optimizadores

Los optimizadores son algoritmos que se usan para calcular una tasa de aprendizaje de forma dinámica, no es un valor estático, sino que varía en función del estado del entrenamiento. Según vaya ejecutando el entrenamiento, puede que la tasa de aprendizaje incremente o puede que decremente.

Si usamos la metáfora de la persona en una montaña, el número de pasos que va a bajar esa persona estará condicionada a distintos criterios. Algunos de esos criterios pueden ser: Cuanto se ha bajado últimamente o como de rápido se ha bajado.

Es decir, la tasa de aprendizaje que se use se irá adaptando en función a distintos criterios. Uno de esos criterios es como de rápido el aprendizaje está yendo respecto a la pérdida de nuestro modelo entre otras.

Dependiendo del optimizador escogido, usarán unas u otros criterios para ir modificando este valor. Los optimizadores más usados son: Adam[37], Adadelta[38], Adagrad[39] o RMSProp[39].

2.5.7. Overfitting

Sabiendo el funcionamiento básico de una red neuronal usando el algoritmo de *forward-pass* y *backpropagation* se puede pensar que si en cada iteración del modelo, la red entrena y aprende nuevos patrones, se puede sacar la conclusión de que realizando un entrenamiento infinito se puede obtener un modelo perfecto cuyo el error se ha minimizado lo máximo posible y por lo tanto los resultados calculados son los esperados. Pero esto no es así, tener una configuración errónea de la red o entrenar a la red más tiempo de lo necesario provoca lo que es conocido como *overfitting*.

El *overfitting* es un problema bien conocido a la hora de entrenar redes neuronales. En español, significa sobreajustado y suele ser causa de usar un entrenamiento que se ha realizado por mucho tiempo, provocando que el modelo se ajuste perfectamente a los datos de entrenamiento y de algún modo “memorice” los datos y por lo tanto, no sepa extrapolar y adaptarse a otros casos. En otras palabras, se sobreajustan las matrices W de todas las capas y así, el modelo se adapta perfectamente a los datos de entrada produciendo que el modelo no sea capaz de estimar un valor correcto cuando se usa un vector de entrada que antes no lo había visto.

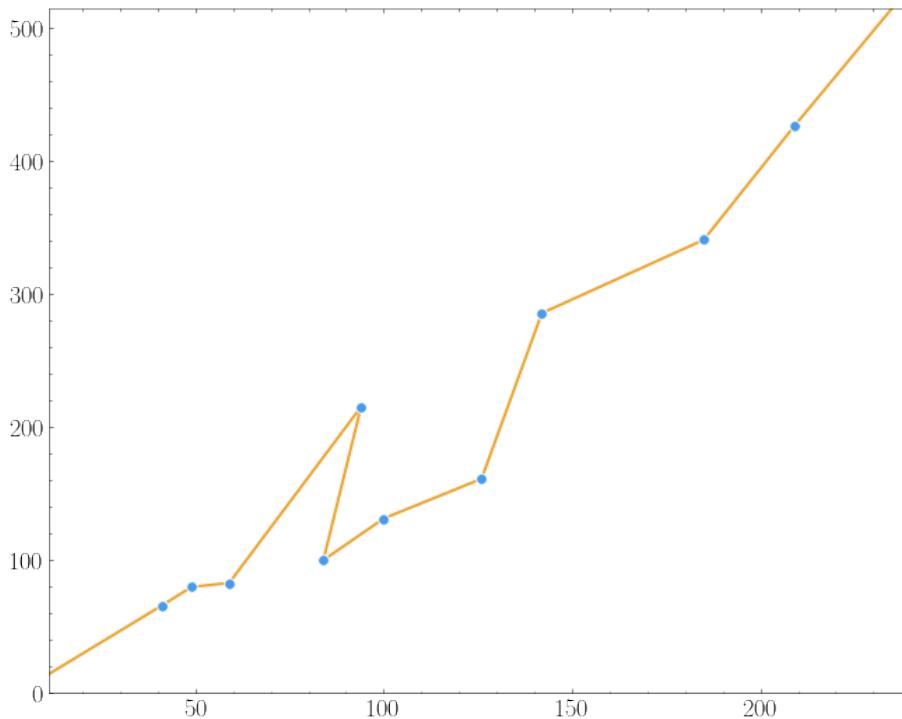


Figura 25: Ejemplo de *overfitting* en una regresión lineal.

manteniendo la precisión de los datos de prueba mientras nuestra red se entrena.

Una de las maneras más rápidas de saber si un modelo se está entrenando con *overfitting* es usando un segundo dataset además del de entrenamiento. Normalmente, antes de crear una red neuronal hay que realizar un preprocessado de datos explicadas en la Sección TODO. Como bien se explica en esa sección, el dataset original se suele dividir en tres dataset distintos: entrenamiento, validación y test. Obviamente, el *dataset* de entrenamiento es usado para entrenar e ir ajustando los parámetros de las matrices W de forma iterativa. Por otro lado, se tiene los *datasets* de validación y test que son usados para evaluar distintas métricas. Si el problema que se quiere resolver con la red es de tipo regresión se suele usar como métrica algún tipo de error explicados en la Sección 2.5.2 como por ejemplo: MAE, MSE o error de *Huber*. Por otro lado si el problema a resolver es de clasificación, se suele usar la precisión como métrica.

Si vemos que la precisión de los datos de prueba ya no mejora, entonces deberíamos dejar de entrenar. Por supuesto, en sentido estricto, esto no es necesariamente un signo de *overfitting*. Podría ser que la precisión de los datos de la prueba y los datos de entrenamiento dejen de mejorar al mismo tiempo. Aún así, la adopción de esta estrategia evitará el exceso de adaptación.

Al finalizar de cada *epoch*, se usarán un subconjunto de datos del *dataset* de entrenamiento y otro subconjunto del *dataset* de validación. Con ellos, se calculará el valor de la métrica seleccionada y se obtendrán dos valores. Estos valores se puede comparar entre ellos. Dos valores que son parecidos indican que el modelo es capaz de estrapolar a

otros casos que no ha usado para el entrenamiento. Por el contrario, si el valor asociado al *dataset* de entrenamiento es mucho mejor que el asociado al de validación esto puede significar que el modelo pueda estar sufriendo de *overfitting*. Un entrenamiento llevado a cabo sin *overfitting* se puede visualizar en la Figura 26.

Al inicio del entrenamiento, como la red no ha sido entrenada y los parámetros inicializados aleatoriamente, el modelo tendrá unas métricas bastante malas, es decir, si se está usando algún error, dicho valor será muy alto. Por otro lado si se está midiendo la precisión del modelo dicho valor distará mucho del 100 %. Estas métricas serán igualmente malas tanto para ambos *datasets*. Según vayan ocurriendo *epochs* en el entrenamiento, llegará un momento que el modelo no pueda mejorar más y comenzará a memorizar los datos con los que está siendo entrenado del *dataset* de entrenamiento provocando que la métrica asociada al entrenamiento sea casi perfecta y la de validación progresivamente vaya empeorando. Un ejemplo de un entrenamiento que sufre de *overfitting* puede verse de la siguiente manera:

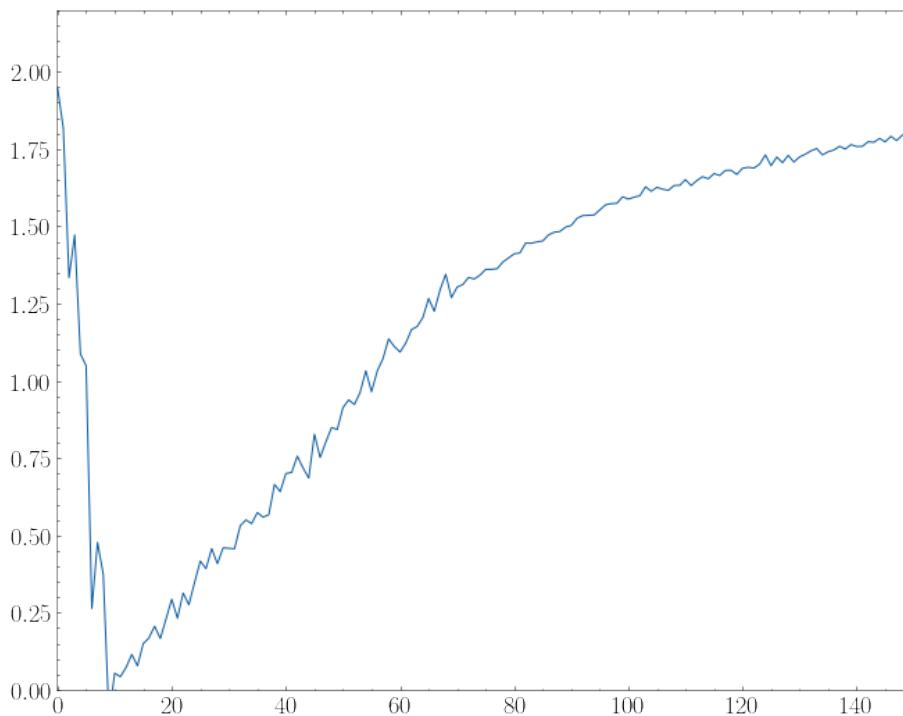


Figura 26: Ejemplo de *overfitting* en una regresión lineal usando un error como métrica.

Al comienzo del entrenamiento ambas métricas van en paralelo, hasta que llegan a la *epoch* 10 aproximadamente, en ese punto se puede visualizar como la métrica de validación poco a poco va empeorando. Es en ese punto donde se ve que la red tiene un problema de *overfitting*. Dependiendo de la métrica usada, el *overfitting* puede aparecer antes o después, pero es un tema más complejo que no se explicará en este trabajo.

Aún así hay varias técnicas que se explicarán a continuación para evitar el *overfitting*:

- Regularización L1 y L2: Son dos métodos que calculan un valor conocido como penalización que se añade al error y de ese modo poder penalizar parámetros con valores grandes. Si una neurona tiene valores grandes puede ser signo de que la neurona está intentando memorizar. De hecho, se considera una mala práctica dejar que un conjunto pequeño de neuronas de la red sean las que más responsabilidad tengan, es decir, que sus parámetros asociados sean muy grandes.

Por un lado, se tiene L1 que es la suma de todos los valores absolutos de w y b . Es una penalización lineal ya que la función asociada es directamente proporcional a los parámetros. La penalización L2 es la suma de todos los parámetros w y b al cuadrado. Es una función no lineal y penaliza con mayor intensidad a los valores grande a diferencia de L1 que penaliza mucho más en proporción a los valores pequeños por ser lineal. Esto causa que el modelo empieza a ser invariante a pequeños valores de entrada y variante solo a los valores grandes. Por ello, L1 es raramente usado a no ser que sea en combinación con L2. Ambas funciones están en función de un valor λ , con este valor se puede dictar cuánto impacto tendrá L1 y L2 en el error final. Matemáticamente:

$$L_{1w} = \lambda \sum_m |w_m| \quad L_{2w} = \lambda \sum_m w_m^2 \quad (80)$$

$$L_{1b} = \lambda \sum_n |b_n| \quad L_{2b} = \lambda \sum_n b_n^2 \quad (81)$$

El error final viene dado por la simple suma del error calculado por la función de coste y L1 con L2:

$$c_T = c + L_{1w} + L_{1b} + L_{2w} + L_{2b} \quad (82)$$

Como se usa la regularización para el cálculo de c , hace falta saber su derivada para poder usar el *back-propagation*. Quedando la derivada de coste de la siguiente forma:

$$\frac{\partial c_T}{\partial w^{L_i}} = c' + L'_{1w} + L'_{1b} + L'_{2w} + L'_{2b} \quad (83)$$

Las derivadas de L1 y L2 son las siguientes:

$$L'_{1w} = \lambda \begin{cases} 1, & \text{si } w_m > 1 \\ -1, & \text{si } w_m < 1 \end{cases} \quad L'_{2w} = 2\lambda w_m \quad (84)$$

$$L'_{1b} = \lambda \begin{cases} 1, & \text{si } b_n > 1 \\ -1, & \text{si } b_n < 1 \end{cases} \quad L'_{2b} = 2\lambda b_n \quad (85)$$

- *Dropout* : Usando esta técnica, en cada iteración del entrenamiento la red se modifica. Supongamos que se tiene un *input* x y el valor deseado y . Ordinariamente, se

entrenaría mediante la propagación hacia adelante de x a través de la red, y luego la propagación hacia atrás para determinar la contribución al gradiente. Con *dropout*, este proceso se modifica. Se comienza eliminando aleatoriamente (y temporalmente) un porcentaje de las neuronas ocultas en la red, mientras que se deja las neuronas de entrada y salida sin modificar. Las neuronas que han sido borradas, serán neuronas "fantasmas" para ese *epoch*:

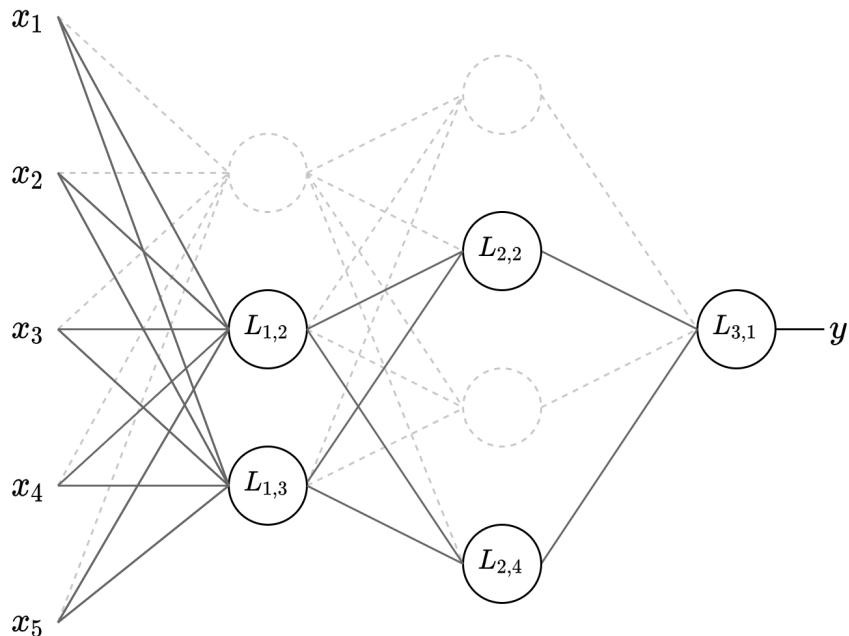


Figura 27: *Dropout* aplicado a una red con un 50 % de probabilidades

Una vez terminado el proceso de propagación hacia adelante y *backpropagation*, se comenzará un nuevo *epoch* eliminando aleatoriamente un conjunto de neuronas. Es resumen, por cada iteración, se eliminar un subconjunto de las neuronas de acuerdo a un porcentaje dado.

Repetiendo este proceso durante toda la fase de entrenamiento, la red aprenderá unas matrices W que se habrán aprendido en condiciones en las que un subconjunto de las neuronas ocultas fueron eliminadas. Cuando realmente se ejecuta la red completa, significará que el número de neuronas activas será mayor. Por ello, se compensar reduciendo la parte proporcional con las que fueron entradas. Por ejemplo, usando un porcentaje igual a 50, los valores de W serán la mitad de lo que el gradiente haya calculado.

- Selección de una tasa de aprendizaje de forma iterativa: Esta técnica se basa en el estudio de varias tasas de aprendizaje en función del *epoch* del entrenamiento. Se suele dar un conjunto de valores de forma ordenada entre un mínimo y un máximo. Cada valor de este intervalo será una tasa de aprendizaje para distintas *epochs* y se podrá estudiar su comportamiento y ver si el descenso del gradiente es capaz de

converger.

Si es capaz de converger, los resultados obtenidos serán los que se esperan, haciendo que la función de coste se reduzca de forma progresiva. Por el contrario, si la tasa de aprendizaje es muy baja o muy alta, provocará que el descenso del gradiente no pueda converger resultando en que el resultado de la función de coste sea irregular y por lo general empeorando el modelo.

2.6. Redes neuronales recurrentes

2.6.1. Nociones básicas

Las Red neuronal recurrente (RNN) son un tipo de arquitectura de redes que son aplicadas a multitud de problemas sobretodo de Procesado del Lenguaje Natural (NLP) como por ejemplo reconocimiento de voz o traducción de texto, pero también es usado para problemas en el que los datos están ordenados en el tiempo y tienen cierta relación como puede ser el caso de predicción de valores en bolsa o detección de objetos en vídeos.

En resumen, las RNN son redes usadas para trabajar con modelos de datos secuenciales. Un ejemplo usando datos secuenciales puede ser el siguiente: Se muestra una foto con un círculo dibujado y se pregunta hacia que dirección está yendo el círculo. Cualquier respuesta a esta pregunta será aleatoria y sin ningún tipo de criterio. Ahora bien, si se muestran varias fotos y se explica que son fotos que se han tomado previamente a la primera foto y en estas fotos se pueden ver que el círculo sigue una trayectoria, entonces la respuesta a la pregunta será trivial porque fácilmente se puede predecir hacia donde irá el círculo.



Figura 28: Ejemplo gráfico de un dato único vs. una secuencia.

De esta forma se puede ver la importancia de tener información temporal y como esta se puede usar para resolver predicciones. La información secuencial está presente en el mundo de diferentes formas: audio (secuencia de bits), texto(secuencia de caracteres o de palabras) o valores que se modifican a lo largo del tiempo.

La forma en que las RNN funcionan es gracias a un concepto que se conoce como memoria secuencial. La memoria secuencial es la capacidad para reproducir secuencias de palabras, números, letras, símbolos... de un humano. Por esa razón, es fácil de decir el abecedario y, sin embargo, es más difícil decir el abecedario hacia atrás.

Usando algún tipo de bucle dentro de la red se puede emular esta memoria secuencial para que una arquitectura de red pueda usar información previa y así poder procesar el

cálculo de un nuevo valor. Esta información que es traspasada usando el bucle es lo que se conoce como estado oculto (h , *hidden state* en inglés) y contiene información de los valores de entrada previos.

Cuando se hace referencia a una RNN, se está hablando de una red que al menos una de las capas es recurrente y por lo tanto tiene algún tipo de bucle. Es decir, pueden existir una RNN con una sola capa recurrente o bien una RNN con dos capas recurrentes o cualquier tipo de combinación con al menos una capa recurrente. Se muestra a continuación varios ejemplos:

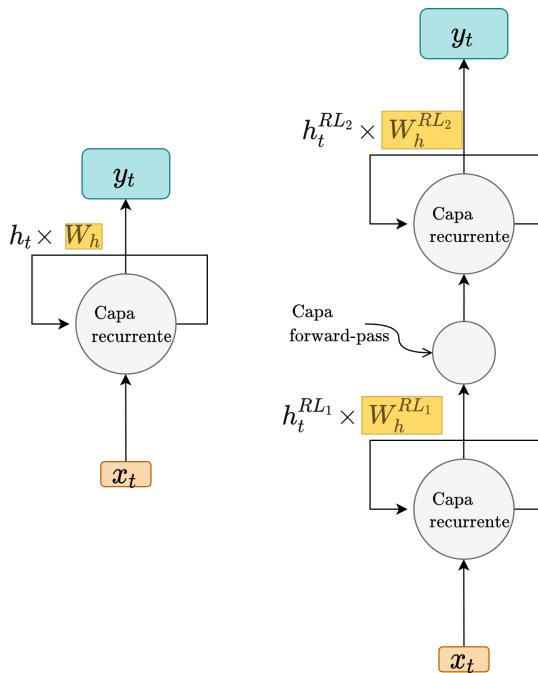


Figura 29: Varios ejemplos de RNN

A partir de ahora, en vez de representar el bucle con una conexión reflexiva, se hará usando el eje horizontal para cada una de las iteración y de forma que queda una cadena de redes enlazadas. Esta es la forma más común de ver las RNN representadas.

En el anterior diagrama, se ha introducido una nueva matriz: W_h . Esta matriz es la matriz propia de pesos para el vector h . Además, como puede haber varias capas recurrentes, para diferenciar W_h y h se usará el índice RL_i siendo i el número de la capa recurrente.

2.6.2. Tipos

Hay tres tipos distintos de RNN:

- Muchos a uno: A partir de una secuencia generar un vector. Por ejemplo, valorar del uno al diez opiniones sobre productos escritas por clientes.

- Muchos a muchos: A partir de secuencia generar una secuencia. Por ejemplo, traducir texto de un idioma a otro.
- Uno a muchos: A partir de un solo vector se genera una secuencia. Por ejemplo, una red especializada en describir los elementos de una imagen.

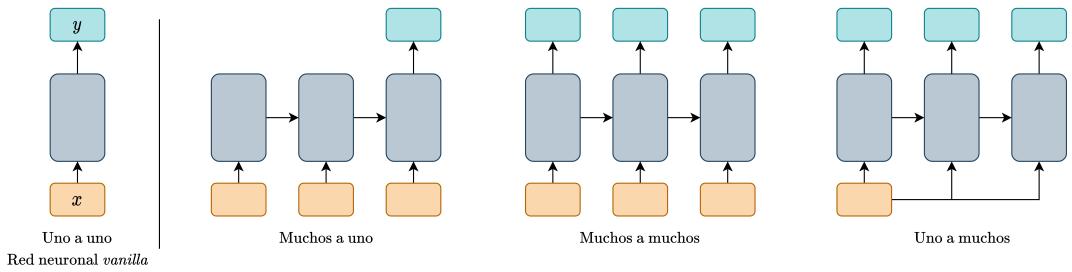


Figura 30: Representación de funcionamiento de los tipos de RNN

2.6.3. Algoritmo *forward pass* en RNN

El funcionamiento del *forward pass* en una RNN es parecido a una capa normal. Una capa recurrente básica suele tener una única capa con la función de activación como puede ser *tanh*. El vector h_t es un vector calculado por esta función de activación pero que no solo recibe como parámetro x , sino que también recibe el estado oculto anterior h_{t-1} . De la misma manera, el vector h_{t+1} tendrá información tanto de h_t y de x_{t+1} . Como se puede ver, el estado oculto de cada capa va pasando de capa en capa. Gráficamente:

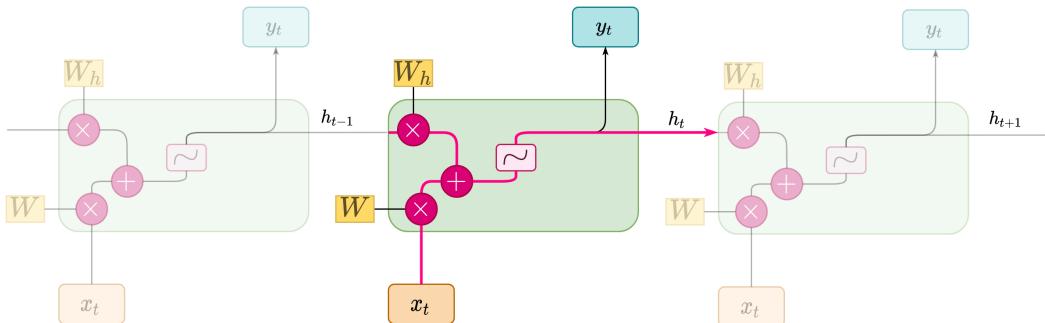


Figura 31: Funcionamiento de una RNN

Por cada iteración la presencia de información de anteriores h va disminuyendo provocando el desvanecimiento del gradiente, concepto que se explicará posteriormente. Este flujo de información se puede ver mejor con el siguiente diagrama:

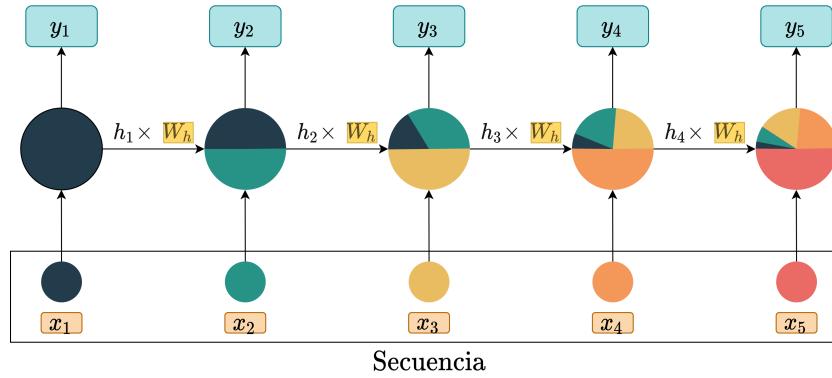


Figura 32: Estructura de una capa recurrente

Las capas recurrentes al igual que ocurre con las capas básicas de una red, tendrán asociado una matriz de pesos (aunque no tienen *bias* b). Esta matriz será representada con W_h dejando W para matrices de pesos de capas básicas. Pueden existir múltiples capas recurrentes cada una con una matriz W_h independiente cada una, por eso se hace uso del índice RL_i como se ha explicado en la Figura 29.

El cálculo de y es parecido al cálculo que se realiza en una red neuronal normal. Partiendo de la ecuación explicada en la Ecuación 35:

$$y = a(W^{L_l} \cdot (a(W^{L_{l-1}} \cdot \dots \cdot (a(W^{L_1} \cdot x')^{L_1})^{\dots L_{l-1}}))^{L_l}) \quad (86)$$

Simplificando este cálculo a una sola capa:

$$\begin{aligned} z &= W^{L_l} \cdot y^{L_i} \\ y &= a(z) \end{aligned} \quad (87)$$

Si se trata de una capa recurrente, se ha de añadir un nuevo sumando a la operación, pero se quiere que siga teniendo las mismas propiedades de la función de activación, por lo que en realidad, lo que se va a modificar es el cálculo de z :

$$\begin{aligned} z &= W^{L_l} \cdot y^{L_i} + W_h \cdot h \\ y &= a(z) \end{aligned} \quad (88)$$

Por ejemplo si se tiene una red recurrente formada por cinco capas, donde la primera y la tercera son capas recurrentes, el cálculo de y final es el siguiente:

$$y = a(W^{L_5} \cdot a(W^{L_4} \cdot a(W^{L_3} \cdot a(W^{L_2} \cdot a(W^{L_1} \cdot x' + W_h^{RL_1} \cdot h_1))^{L_2} + W_h^{RL_2} \cdot h_2)^{L_3})^{L_4})^{L_5} \quad (89)$$

El cálculo de z dependerá de si la capa recurrente es la primera:

$$z^{L_1} = W^{L_1} \cdot x' + W_h \cdot h \quad (90)$$

En otro caso, si la capa recurrente no es la primera:

$$z^{L_i} = W^{L_i} \cdot y^{L_{i-1}} + W_h \cdot h \quad (91)$$

2.6.4. Algoritmo *backpropagation* y descenso del gradiente en RNN

El cálculo del error en este tipo de arquitecturas es igual que en las redes *forward-pass*, simplemente se aplica cualquier función de coste usando como dos argumentos: el valor predicho y el valor real como se explica en la sección 2.5.2. Todos los errores calculados serán sumados y guardados en C . Este error se usará para calcular el gradiente de todas las neuronas. Para ello hará falta retropropagar el error. Este error se retropropagará de la misma forma que se ha explicado en la sección 2.5.4 para las capas que no sean recurrentes.

Para las capas que apliquen algún tipo de bucle, se le debe de añadir una nueva derivada. Para retropropagar el error, hace falta dividir dicho error por partes proporcionales para cada neuronas. Si al añadir el bucle, se está añadiendo una nueva conexión consigo misma, el error se tendrá que dividir en dos partes: Una parte para las capas anteriores y otra para esa misma capa. A continuación se puede ver un diagrama representando como se retropropaga el error. En resumen, el error se retropropaga tanto en la red como en el tiempo, por eso a esta variante se le llama *backpropagation* a través del tiempo.

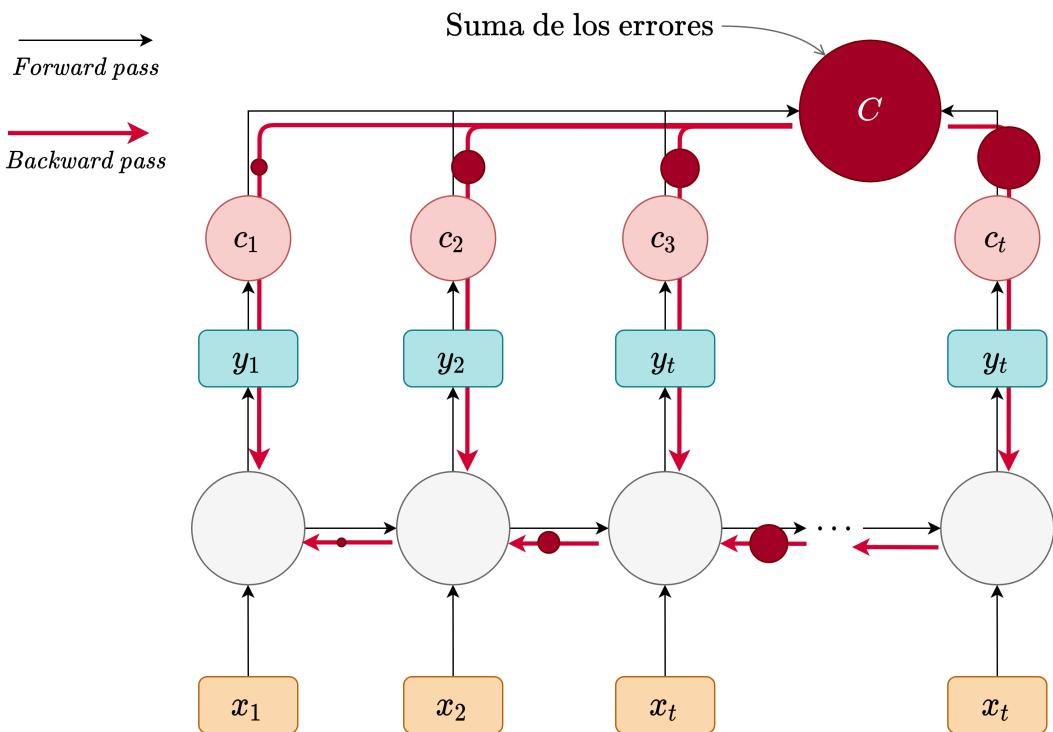


Figura 33: *Backpropagation* a través del tiempo en RNN

Como se puede ver, el error calculado C a partir de todas las c_i , fluyen hacia atrás en el tiempo. Para la primera parte del cálculo del *backpropagation* se pueden usar las mismas ecuaciones 67 y 66. En estas ecuaciones se realiza el cálculo del coste respecto a w y respecto a b . El coste debe de ser C y no c_i . Para las capas recurrentes, habrá que

calcular una nueva derivada: $\frac{\partial c}{\partial W_H}$. Aplicando la *chain-rule* como se ha explicado en la sección 2.5.4.

$$\frac{\partial C}{\partial W_h} = \frac{\partial C}{\partial z^{L_i}} \cdot \frac{\partial z^{L_i}}{\partial W_h} \quad (92)$$

Y al igual que en el cálculo del gradiente que se ha realizado para w y para b en la ecuación 55, se tiene que calcular el gradiente de W_H de la misma forma:

$$\nabla f_{w_h} = \left(\frac{\partial C}{\partial w_h^{RL_1}}, \frac{\partial C}{\partial w_h^{RL_2}}, \dots, \frac{\partial C}{\partial w_h^{RL_n}} \right) \quad (93)$$

Un detalle a recalcar es que se está usando w_h y no W_H , puesto que esta última hace referencia a la matriz de pesos de la capa recurrente. El gradiente se debe de calcular de forma independiente por cada neurona, al igual que ocurre con w y W . Este concepto se explica con más detalle en la Sección 2.4.4. Una vez calculado todos ∇f_h para cada neurona, se puede actualizar la matriz de pesos w_H :

$$H^* = H - \eta * \nabla f_{w_h} \quad (94)$$

Es importante primero realizar el cálculo del *backpropagation* y posteriormente realizar el cálculo del gradiente.

2.6.5. Problemas del descenso del gradiente en capas recurrentes "vanilla"

como se puede observar en la Figura 32, por cada paso de tiempo, hay que multiplicar la matriz W_H por H . La actualización de la capa recurrente viene dado por una función de activación no lineal. Eso provoca que el cálculo del gradiente, que es la derivada del coste respecto a los parámetros que forman la red, incluyendo todos los estados iniciales. Requiere muchas multiplicaciones repetidas de estos pesos y también el uso repetido de las derivadas de la función de activación. Y esto puede ser problemático:

- **Gradiente explosivo:** Ocurre cuando muchos de los valores involucrados en el proceso de multiplicación repetitiva de matrices son mayores a 1 provocando que por cada iteración, el gradiente sea cada vez más grande y por lo tanto la actualización de los parámetros: w^* , b^* y w_H^* tienda al infinito. Hay una solución a este problema y es aplicar lo que se conoce como recorte del gradiente que básicamente normaliza los valores para que no sean tan grandes.
- **Desvanecimiento del gradiente:** Ocurre cuando muchos de los valores involucrados en el proceso de multiplicación repetitiva de matrices son pequeños provocando que por cada iteración, el gradiente sea cada vez más pequeño y por lo tanto la actualización de los parámetros w^* , b^* y w_H^* tienda a 0. Este es uno de los principales problemas al entrenar redes recurrentes.

Se puede ver un ejemplo de este problema si se selecciona un número del 0 a 1 e iterativamente se va multiplicando por sí mismo. Se puede ver que este valor por cada iteración se hace más y más pequeño y eventualmente, se desvanecerá. Esto provoca que el error sea mucho más difícil de propagar más lejos en el pasado y de esa forma se estaría programando una red que aprendiera pasado cercano. Esto no significa que sea un problema, puede haber en alguna ocasión que se necesite de este tipo de arquitecturas. Gráficamente, este desvanecimiento de información se puede ver en la Figura 33, donde por cada iteración se tiene la información de x actual y parte de información de las anteriores iteraciones.

Este error se puede evitar si:

- Eligiendo ReLU como función de activación: Esto ayuda a que el valor de la derivada de $a()$ no se haga pequeña pero solo cuando $x > 0$.
- Inicializando los parámetros de forma no aleatoria: Por ejemplo inicializando la matriz con la matriz identidad.
- Diseñando la arquitectura de la red de forma óptima: Usar una capa más compleja como puede ser LSTM, GRU... que es más efectiva a la hora de rastrear memorias a largo plazo controlando qué información pasa y cuál es usada para actualizar su estado interno. Específicamente es el concepto de una puerta

2.6.6. Capas LSTM

Las *Long-Short Term Memory* (LSTM) (Memorias a corto-largo plazo en español), son las capas recurrentes más usadas por su capacidad de mantener dependencias a largo plazo y son usadas extensamente en comunidades de aprendizaje profundo.

La clave detrás de las LSTM son las puertas que permite a la neurona añadir o eliminar activamente información. Está formado por varias funciones de activación σ y una $tanh$. La función de activación σ tiene como consecuencia que el valor de salida sea un valor entre 0 y 1. Intuitivamente se puede pensar en que estas funciones tienen como objetivo capturar y retener información entre 0, o nada, y 1, todo.

A continuación se puede ver un diagrama de cómo funciona una puerta LSTM.

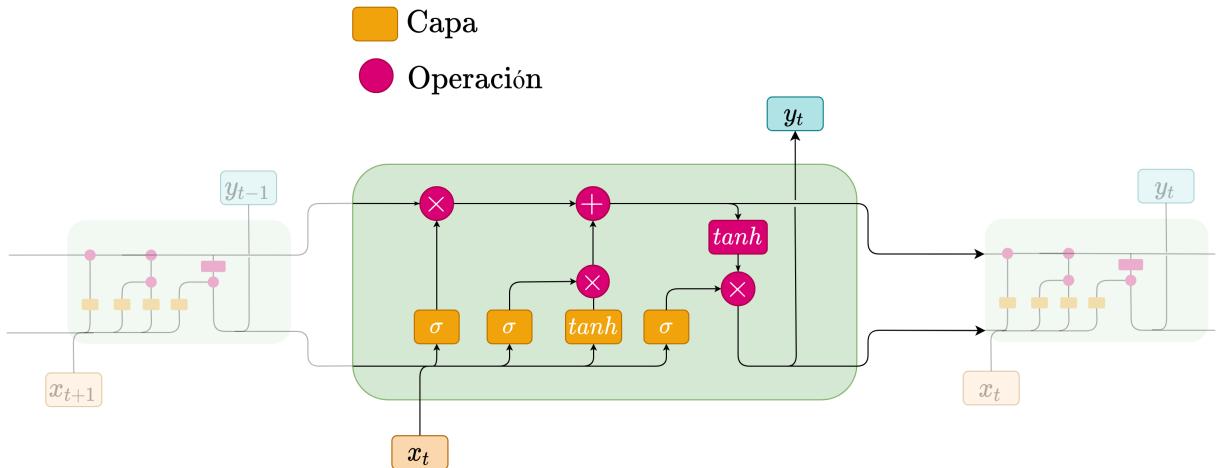


Figura 34: Funcionamiento de una LSTM

Las LSTM se forman básicamente por cuatro pasos distintos:

1. Olvidar la información irrelevante: En la primera salida de Esta parte es la primera función σ junto a la operación del producto escalar entre la salida de σ y el estado anterior h_{t-1} .
2. Guardar nueva información: Está formada por la segunda función σ y la función $tanh$ al lado suya.
3. Usar los dos pasos anteriores para calcular un nuevo estado interno h_t : Es la parte representada por la línea superior y calcular c_t . Suele considerarse a este vector como la memoria de LSTM.
4. Computar el valor de salida y_t : Formada por la última función σ y por la función $tanh$. Controla qué información es importante y guarda la información sobre el estado actual.

Con este diseño de puertas se puede diseñar una neurona recursiva que permita el flujo de los gradientes de forma ininterrumpida y evitando su desvanecimiento.

En resumen, una neurona LSTM permite separar en dos partes tanto la memoria como el estado interno permitiendo así que la red pueda aprender dependencias a largo plazo en una secuencia. Esto lo hace mediante el uso de puertas para controlar el flujo de información: olvidando información irrelevante, guardando información importante, actualizando el estado interno de forma selectiva y guardando parte de la información. Todo ello permitiendo que el flujo de los gradientes sea inalterado para evitar el desvanecimiento del gradiente.

2.7. Trabajos y proyectos similares

En esta sección se analizarán varios proyectos, estudiando tanto su implementación como sus resultados obtenidos. Los resultados de estos trabajos nos han sido útiles para comparar nuestros modelos y resultados.

2.7.1. *Development of a station-level demand prediction and visualization tool to support bike-sharing systems' operators*

En este trabajo [40], se crea un proceso de predicción de la demanda de bicicletas y una herramienta de visualización de estas para la ciudad de Thessaloniki, Grecia. Se usan diferentes modelos en los que se encuentran *XGBoost*, *Random Forest* y redes neuronales entre otros. Los datos de los que dispone son lo siguientes:

- Información geográfica: Coordenadas de la estación, usando por lo tanto dos atributos: latitud y longitud.
- Información temporal expresada en: Hora, día de la semana, mes y año. Las tres primeras, son mapeadas a una tupla (*seno* y *coseno*).
- Información sobre el calendario laboral: Usando dos variables: una para identificar si es día laboral o no y otra para identificar si es domingo o no.
- Información meteorológica: Usando tanto la información meteorológica del momento como el pronóstico a corto plazo. Este tipo de información incluye: temperatura media, velocidad del viento, cantidad de nubes y precipitación.

Para la red neuronal, se ha usado una red densa *forward-pass* con 2 capas ocultas de 20 y 8 neuronas respectivamente y una capa de salida de una sola neurona. Para las métricas han usado: MAE, MSE, RMSE y RMSLE. El *dataset* comprende los años 2014 al 2018 y se desechan los datos comprendidos entre las 00:00 y las 6:00 justificando que la mayoría de estaciones no están automatizadas y operan solo durante el resto del tiempo. Sus resultados son los siguientes:

Métrica	Valor
MAE	0.89
MSE	2.66
RMSE	0.64
RMSLE	0.49

Este proyecto es una muestra de un buen caso de uso para una red neuronal, usando las predicciones para mostrarlos en una herramienta web y de esa forma que los usuarios y la empresa puedan tomar decisiones. En cuanto a lo que es la red neuronal considero que se podría haber hecho mucho mejor porque la red usada es muy simple y los hyperparámetros usados son los por defecto de *sci-kit learn* y los resultados son regulares y mejorables.

2.7.2. *Predicting station-level hourly demand in a large-scale bike-sharing network: A graph convolutional neural network approach*

Este estudio propone un modelo de red neural convolucional que puede aprender correlaciones heterogéneas ocultas por pares entre estaciones para predecir la demanda horaria a nivel de estación a gran escala.

Se exploran dos arquitecturas del modelo GCNN-DDGF: El modelo CNN-DDGF que contiene los bloques de convolución y de avance, y GCNNrec-DDGF contiene además un bloque LSTM para capturar las dependencias temporales en la serie de demanda. Además, se proponen cuatro tipos de modelos GCNN cuyas matrices de adyacencia se basan en diversos datos del sistema de bicicletas compartidas, entre ellos la matriz de distancia espacial (SD), la matriz de demanda (DE), la matriz de duración media del viaje (ATD) y la matriz de correlación de la demanda (DC). Estos seis tipos de modelos de GCNN y otros siete modelos de referencia se construyen y comparan en un conjunto de datos de Citi Bike de la ciudad de Nueva York que incluye 272 estaciones y más de 28 millones de transacciones de 2013 a 2016.

Los resultados muestran que el GCNNrecDDGF tiene el mejor rendimiento en términos del RMSE, el MAE y el coeficiente de determinación (R^2), seguido por el GCNNreg-DDGF. Superan a los otros modelos. Se encuentra para capturar alguna información similar a los detalles incrustados en las matrices SD, DE y DC. Y lo que es más importante, también descubre correlaciones heterogéneas ocultas por pares entre estaciones que no son reveladas por ninguna de esas matrices.

Métrica	Valor
MAE	1.44
RMSE	2.46
R^2	0.67

Este proyecto es una muestra de un buen caso de uso para una red neuronal, usando las predicciones para mostrarlos en una herramienta web y de esa forma que los usuarios y la empresa puedan tomar decisiones. En cuanto a lo que es la red neuronal considero que se podría haber hecho mucho mejor porque la red usada es muy simple y los hiperparámetros usados son los por defecto de *sci-kit learn* y los resultados son regulares y mejorables.

2.7.3. Competición de Kaggle: *Shared bikes demand forecasting*

La plataforma *Kaggle* es una plataforma donde los interesados en Ciencia de Datos pueden compartir blogs, conocimientos y *datasets* entre otros recursos. Además, la propia plataforma o empresas que quieren contratar nuevo personal que sea experta en Ciencia de Datos realizan competiciones frecuentemente donde cualquier persona o equipo del mundo puede participar. Es por ello, que es común ver múltiples competiciones activas en

Kaggle con diferentes problemas realcionados con *Big Data* y sus *datasets*.

Durante el verano de 2020, tuvo lugar una competición [41] que trataba de resolver el mismo problema que se ha planteado en este presente trabajo: predecir la cantidad de viajes que habrá desde cada una de las estaciones en una ciudad. En este caso, la competición usó un *dataset* que incluía: ciudad (había multiples ciudades), tiempo en horas, cantidad de bicicletas en una hora, multitud de variables en relación con la meteorología(20 variables) y un calendario laboral asociado.

La métrica que se usó para medir los modelos que entregaban los participantes fue la *RMSLE* y los mejores resultados obtenidos fueron los siguientes:

Métrica	RMSLE
Guillermo Alejandro Chacon	0.18721
Maria Garcia	0.18973
Eleanor Manley	0.19837
Milan Goetz	0.20110
Luisa Runge	0.20162
Paula San Roman Bueno	0.20209
Diego Cuartas	0.20498
Valentina Premoli	0.20569

3. DESARROLLO

3.1. Definición del problema

Este trabajo tiene como objetivo el estudio de las redes neuronales recurrentes. No obstante, no solo se centra en el apartado teórico sino que también en el apartado práctico. Por ello, se ha buscado un problema para poder tratar de resolver como ejemplo. El problema que se ha tratado de resolver es la predicción de uso de bicicletas compartidas (*bikesharing*) en la ciudad de Chicago. Como se quiere llevar a cabo una solución usando aprendizaje supervisado, se necesitan previamente una cantidad enorme de datos. Se ha usado Chicago justo por ese motivo, por la facilidad de acceso a los datos públicos necesarios para poder entrenar a la red sin ningún tipo de restricción ni licencia. En concreto se ha descargado el dataset ofrecido por la compañía *Divvy* [42].

Además, el ayuntamiento de Chicago ofrece mapas interactivos [43] que han sido de gran ayuda para poder estudiar las distintas propiedades de las estaciones al igual que hay multitud de proveedores de datos tanto meteorológicos [44] como de otro tipo que también se pueden añadir al *dataset* y así mejorar la precisión del modelo pero que por falta de tiempo y por los resultados obtenidos no se ha podido realizar como se explica en la Sección 8.

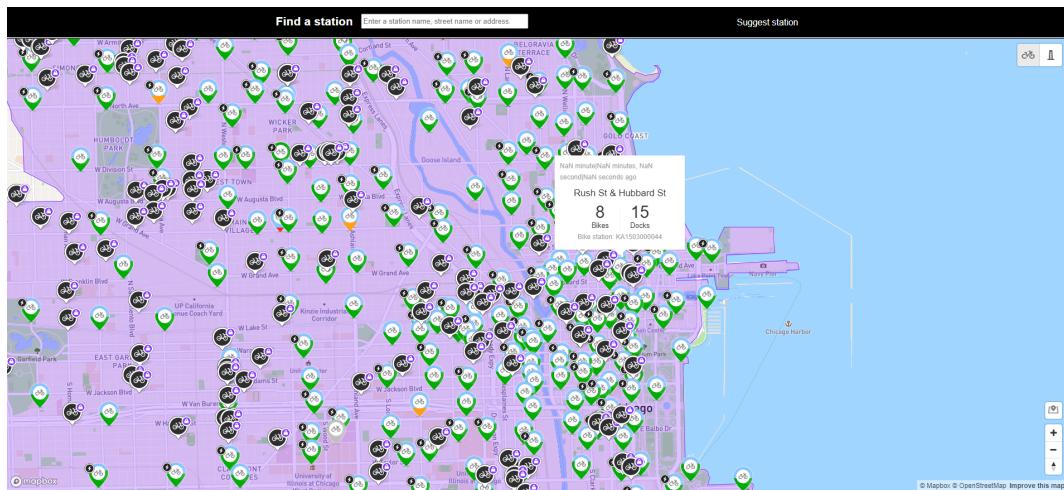


Figura 35: Mapa interactivo de la red de estaciones de alquiler de biciletas en Chicago [43]

3.2. Herramientas y librerías usadas

El lenguaje usado para el desarrollo de esta librería ha sido Python 3.6 [45]. Este lenguaje es una opción de *facto* junto a *R* en el campo Inteligencia Artificial. Junto con Python, se han desarrollado un conjunto de librerías que favorecen que este lenguaje sea la opción favorita para desarrollar proyectos de este tipo. Las librerías usadas en específico para este proyecto han sido:

- Tensorflow [31]: es la plataforma de Aprendizaje profundo más importante del mundo. Esta librería *open-source* de Google va más allá de la Inteligencia Artificial, pero su flexibilidad y gran comunidad de desarrolladores lo ha posicionado como la herramienta líder en el sector del Aprendizaje profundo.
- Keras [46]: Keras es una biblioteca de código abierto escrita en Python, que se basa principalmente en el trabajo de François Chollet, un desarrollador de Google, en el marco del proyecto ONEIROS. El objetivo de la biblioteca es acelerar la creación de redes neuronales: para ello, Keras no funciona como un framework independiente, sino como una interfaz de uso intuitivo (API) que permite acceder a varios frameworks de aprendizaje automático y desarrollarlos. Entre los frameworks compatibles con *Keras*, se incluyen TensorFlow.
- Numpy [47]: Numpy es la librería por excelencia para computación científica en Python. Trae integradas muchas funciones de cálculo matricial de N dimensiones, así como la transformada de Fourier, múltiples funciones de álgebra lineal y varias funciones de aleatoriedad.
- Pandas [48]: Es una biblioteca de código abierto que proporciona estructuras de datos y herramientas de análisis de datos de alto rendimiento y fáciles de usar para el lenguaje de programación Python.
- Matplotlib [49]: Es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática *NumPy*. Proporciona una API, pylab, diseñada para recordar a la de MATLAB. Se ha usado el tema gráfico llamado SciencePlots [50].

3.3. Preprocesado de datos

Como bien se ha dicho en la anterior sección, se ha usado un *dataset* público ofrecido por la compañía *Divvy* [42]. Este *dataset* contiene el historial con todos los viajes registrados desde el 2004 al 2019 en la ciudad de Chicago. Estos datos se ofrecen divididos por partes: o bien por cuatrimestres o bien por meses. El *dataset* final contiene 26.328.986 de viajes en toda la red con un total de 633 estaciones. Por cada viaje, se tienen 12 etiquetas asociados a ellos. En el Apéndice B se puede ver una muestra de los datos recogidos en él.

Para el desarrollo de este proyecto, se han creado cuatro módulos distintos. Cada una de estos módulos tendrá como resultado una estructura de datos guardada en CSV a excepción del módulo del generador de ventanas que contendrá información que será útil para el siguiente módulo. El principal motivo de que los dos primeros módulos tengan como resultado la creación de un fichero es para ahorrar tiempo de cómputo innecesario cada vez que se quiera entrenar un modelo. Como para entrenar a los modelos es necesario multitud de pruebas con diferentes configuraciones, es un proceso que se hubiera repetido continuamente y cuyo resultado siempre hubiese sido el mismo. Por otro lado, el generador de ventanas, módulo que se explicará más adelante, al ser un módulo que es

instantáneo y que además genera distintos resultados en función del modelo que se quiera desarrollar no guarda el resultado en algún archivo.



Figura 36: Estructura del proyecto divida por módulos.

Durante las siguientes secciones se explicarán cada uno de estos módulos en detalle incluyendo diagramas y código.

3.3.1. *Feature selection - Unión de datasets y filtro de variables*

El *dataset* original [42] está divido en años y cada año en múltiples partes. Por lo tanto, la primera tarea de todas ha sido la unión de cada una de estas partes en un único fichero. El formato de archivo ofrecido por *Divvy* es en un CSV como es normal, y por ello se ha usado principalmente la función `read_csv` de pandas que convierte un CSV a un DataFrame. Por otro lado, cada parte tenía su propia nomenclatura de columnas por lo que ha sido necesario crear una única nomenclatura para poder trabajar con un único fichero y por ello han sido renombradas algunas columnas. Una vez realizado el renombrado de columnas, el DataFrame será anexionado a otro que contendrá todos los datos y que será el que se usará para los siguientes pasos.

"Un DataFrame es una estructura de datos etiquetada en 2 dimensiones con columnas de tipos potencialmente diferentes. Se puede pensar en ello como una hoja de cálculo o una tabla SQL. Generalmente es el objeto más comúnmente usado en la librería de *pandas*." [48]

Un ejemplo de DataFrame que se ha visto con anterioridad podría ser la tabla 2. Para poder inicializar un DataFrame se puede realizar desde un diccionario de Python, donde cada clave sea el string que identifica a la columna y cada valor sea un vector con la misma cantidad de valores que filas quiere que se tenga. Otra forma de inicializar un DataFrame es mediante el uso de funciones como `read_csv()`, `read_h5()` o `read_pickle()`.

El código usado es algo parecido a lo que se muestra a continuación:

```

import pandas as pd

# Path to the csv files
csv_filenames = ["trips-2014-Q1", "trips-2014-Q2-Q3",
                 "trips-2014-Q4", "trips-2015-Q1-Q2",
                 # ...
                 "trips-2019-Q3-Q4"]
csv_paths = [f"/path/to/{file}.csv" for file in csv_filenames]

# This will be the final DataFrame
  
```

```

df = pd.DataFrame()

# For every CSV do
for path in csv_paths:

    # CSV to DataFrame
    df_temp = pd.read_csv(path)

    # Parse date as datetime which is always first column
    df_temp[cols[0]] = pd.to_datetime(
        df_temp[cols[0]],
        format='%Y-%m-%d %H:%M:%S',
        infer_datetime_format=True)

    # Rename the columns depending on their
    # current names with a external function
    df_temp = rename_columns(df_temp)

    # Merge the DataFrames
    df = pd.concat([df, df_temp], join='outer')

# ...

```

Una vez obtenido la variable `df` que recoge todos los viajes, se necesita realizar un filtro de las columnas. En este punto, el `DataFrame` contiene 12 valores asociados a cada viaje.

```

df.columns

'''
Index(['trip_id', 'start_time', 'end_time', 'bikeid',
       'tripduration', 'from_station_id',
       'from_station_name', 'to_station_id',
       'to_station_name', 'usertype', 'gender',
       'birthyear'],
      dtype='object')
'''
```

De este `DataFrame` solo se seleccionan dos:

- `start_time`: Este dato es necesario porque se quiere predecir el uso de bicicletas a partir de una información temporal. Este valor, dado en segundos usando el formato "Tiempo Unix" [51] que permite identificar en qué fecha y hora se inició el viaje entre otros datos. Gracias a ello, se pueden identificar distintos patrones, ya que estos varían durante la época del año. No existen los mismos patrones durante el invierno que durante el verano o incluso, no es lo mismo las 2 de la madrugada que las 2 de la tarde.
- `from_station_id`: Los modelos que se van a construir predecirán el número de viajes que se inician en cada estación de Chicago de forma independiente. Este valor por tanto, será usado para calcular el número de viajes iniciados en un intervalo en cada estación. Una tarea similar hubiese sido predecir la cantidad de viajes que finalizan en cada estación para un intervalo y de ese modo se podría controlar el flujo de bicicletas por la red de estaciones.

Si ese hubiese sido el caso, hubiese sido necesario hacer uso de `to_station_id` y repetir el mismo proceso que se explica a partir de este punto.

El resto de atributos también son útiles para otro tipos de problemas como por ejemplo predecir el destino más probable en función de la estación donde se inicia el viaje, u obtener que conexiones son las más transitadas. Sin embargo, está fuera de los objetivos que se han planteado para este trabajo por lo que solo se usarán las columnas previamente mencionadas.

El dataset resultante por tanto, es el mismo dataset original pero con solo las columnas `start_time` y `from_station_id`. En el código esto es fácil de implementar añadiendo solo la siguientes líneas:

```
# ...

# Filter the columns that we need
cols = ["start_time", "from_station_id"]
df = df[cols]

# Save the DataFrame in a CSV
df.to_csv("trips.csv", index=False)
```

Se podrían haber usado más columnas que ayudasen a la predicción ajenas a las ya filtradas, sin embargo, por falta de tiempo y porque los resultados obtenidos eran de por sí bastante buenos, no se ha visto la necesidad de añadirlos. Esto se discute más adelante en la sección 8 .

Gráficamente este módulo realiza lo siguiente:

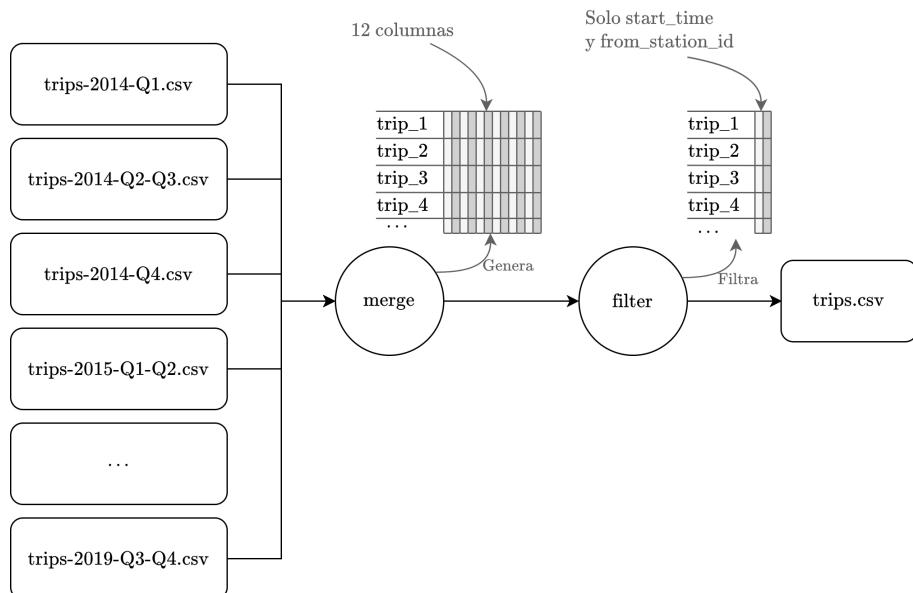


Figura 37: Estructura del módulo *feature-selection*.

3.3.2. Definición de entradas y salidas de las redes neuronales

Antes de continuar explicando los pasos llevados a cabo, se tiene que definir la estructura del *dataset* que se necesita para entrenar a las redes neuronales. Uno de los principales requisitos en el diseño de redes neuronales es la correcta especificación del vector de entrada y salida de la red neuronal. La cantidad de variables y el tipo de variables que se usarán en la red es una de las principales claves para que el modelo funcione de forma satisfactoria. En este apartado se explicará tanto la estructura del vector de entrada como la estructura del vector de salida acompañado de ejemplos gráficos.

Los modelos con los que se trabaja tendrán en el vector de entrada valores que representan la fecha y la hora representados con los siguientes atributos: *hour (h)*, *day_of_week(d)* y *month(m)*. Para ello es necesario que los viajes se agrupen por intervalos, creando de este modo nuevas columnas(*quantity_j*), siendo *j* un entero que represente al identificador de la estación dentro de la red. Estas columnas serán referenciadas con una *q*. En total hay 633 estaciones en la ciudad de Chicago. Por lo que para cada intervalo, la red neuronal contará con 636 valores de entrada. Se pueden ver ejemplos gráficos en la Figura 38 y 39. El tamaño del *dataset* final será de *n_intervals × (n_stations + 3)*.

Lo más básico sería crear un modelo que toma como entrada un único intervalo y predijese un único intervalo representado por el vector de salida. Para este modelo, la entrada de la red sería de 636, formado por $[h, d, m, q_j], j \in [1, 633]$, donde *h*, *d* y *m* son las tres variables que aportan información sobre el intervalo y $q_j, j \in [1, 633]$ son valores que representan la cantidad de viajes iniciados para la estación *j* en dicho intervalo. La salida sería un vector de 633 elementos que representaría la predicción del intervalo inmediatamente posterior al intervalo de entrada y tendría el mismo orden que el vector de entrada $quantity_j, j \in [1, 633]$.

Si se quiere usar más intervalos en el modelo, el vector de entrada tendrá un número de elementos múltiplo de 636. Por ejemplo, si se quieren usar 5 intervalos como vector de entrada, entonces la red tendrá una primera capa con $636 \times 5 = 3180$ unidades o neuronas. De forma similar, si se quiere que la salida de la red sea un vector que contenga la predicción para todas las estaciones de la red, el tamaño del vector resultante será múltiplo de 633.

En los modelos Autoregresivo (AR), a pesar de ser modelos capaces de predecir múltiples intervalos si fuese necesario, la salida de estos modelos siempre serán de un intervalo. Esto es debido a que es necesario calcular todos los intervalos de forma independiente pues el modelo AR usa su propia predicción como vector de entrada. Se puede ver más información sobre esto en la Sección 3.4 y 4.2.4.

A continuación, se pueden ver gráficamente varios ejemplos de distintas configuraciones de red usando diferentes combinaciones para la cantidad de intervalos de entrada y salida:

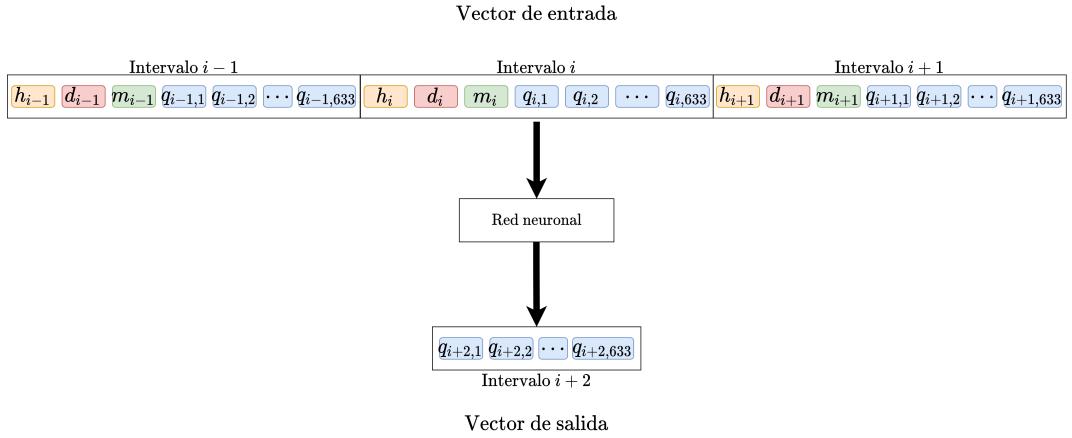


Figura 38: Ejemplo de red neuronal que usa 3 intervalos de entrada y predice 1 intervalo

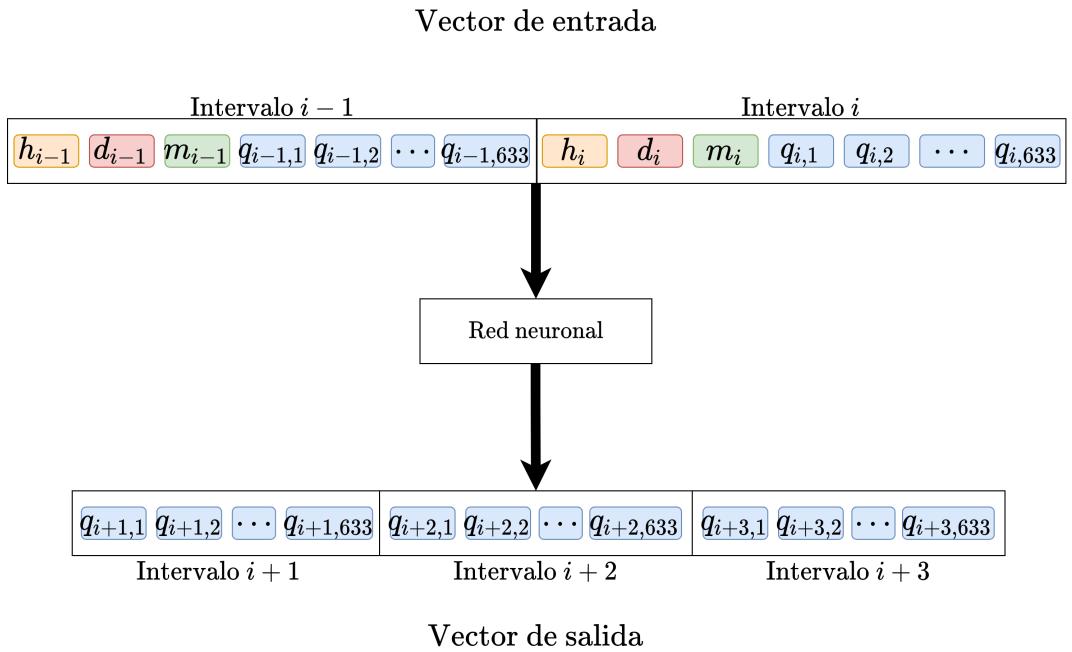


Figura 39: Ejemplo de red neuronal que usa 2 intervalos de entrada y predice 3 intervalos

Esta forma de estructurar los datos y los modelos es muy flexible. Permite que el código se pueda adaptar fácilmente a otros *datasets* y no sea único para la ciudad de Chicago. Además, también permite que se pueda trabajar con un grupo de estaciones más pequeño si se quisiese. Al ser cada columna una estación, simplemente filtrando por las columnas que sean de interés, se podría entrenar los modelos con dichas columnas. También cabría la posibilidad de trabajar por comunidades, agrupando las estaciones en función de una serie de reglas y poder de esa forma estudiar los patrones de un barrio y no de cada estación de forma particular como se realiza en este trabajo.

3.3.3. Modificando el *dataset* para la red neuronal

Una vez explicado el *dataset* que se quiere conseguir junto con la motivación, se explicará en esta sección los pasos seguidos para conseguir dicho resultado. El *dataset* que se tiene en este punto tiene tantas filas como viajes haya habido entre el 2014 y 2019. Por cada fila, se tienen dos atributos: *from_station_id* y *start_time*.

En este módulo se quiere modificar dicho *dataset* para obtener otro que contenga los vectores que puedan ser usados como vectores de entrada en la red. Es decir, se quiere crear un *dataset* que contenga en cada fila los intervalos y en cada columna los datos de dicho intervalo: *hour*, *day_of_week*, *month* y *quantity_j*. Al finalizar todos los pasos de este módulo, los datos se guardarán en un archivo llamado *intervals.csv* y se puede ver una muestra de estos datos en el Apéndice C.

Primeramente, el código de este módulo, carga los datos del anterior módulo del archivo CSV *trips.csv*:

```
import pandas as pd

# Read CSV as DataFrame and use datetime as index
df = pd.read_csv("/path/to/trips.csv", index="start_time")
```

Una vez cargado el DataFrame se han realizado los siguientes pasos:

1. Agrupar los viajes por intervalos: Este paso tiene como objetivo agrupar los viajes que se inician en cada una de las estaciones en intervalos. Se han estudiado diferentes tamaños de intervalos, como por ejemplo, intervalos de 15 ó 30 minutos, pero finalmente, se ha elegido 1 hora porque se reduce de forma considerable la cantidad de filas y sigue siendo un intervalo válido y no muy amplio.

Se han agrupado los distintos viajes en intervalos en función de la columna *from_station_id* y *start_time*. Como ejemplo, suponga que en la variable *df* se tiene el siguiente DataFrame:

<i>start_time</i>	<i>from_station_id</i>
10:10 - 12/2/2018	48
10:28 - 12/2/2018	15
10:56 - 12/2/2018	15
11:03 - 15/2/2018	15
11:12 - 15/2/2018	48
11:15 - 15/2/2018	15
11:18 - 15/2/2018	15
11:31 - 15/2/2018	15
11:44 - 15/2/2018	48
11:49 - 15/2/2018	15
22:00 - 16/2/2018	15
22:15 - 16/2/2018	48
22:37 - 16/2/2018	193
22:56 - 16/2/2018	48

Tabla 3: Ejemplo de *dataset* guardado en *trips.csv*

El código usado para poder agrupar los viajes por intervalos y por estación ha sido el siguiente:

```
INTERVAL = "1H" # It could be also 15Min

df = df.groupby('from_station_id') \
    .resample(INTERVAL, on='start_time') \
    .size() \ # Resampling using the sum rule
    .to_frame() # Converts it to DataFrame
```

Tras ejecutar estas líneas de código y usando el ejemplo de la tabla 3 se puede observar que el resultado es el siguiente:

start_time	from_station_id	quantity
10:00 - 12/2/2018	15	2
10:00 - 12/2/2018	48	1
10:00 - 12/2/2018	193	0
11:00 - 15/2/2018	15	5
11:00 - 15/2/2018	48	2
11:00 - 15/2/2018	193	0
22:00 - 16/2/2018	15	1
22:00 - 16/2/2018	48	2
22:00 - 16/2/2018	193	1

Tabla 4: Ejemplo del *dataset* agrupados por estaciones y por intervalos.

2. Pivolar por intervalos: Este paso tiene como objetivo agrupar todos los intervalos en uno solo, aumentando por tanto la cantidad de columnas y reduciendo la cantidad de filas. Es decir, se quiere tener que cada fila represente un intervalo y cada columna represente una estación. Usando el mismo ejemplo del paso anterior, el DataFrame final quedaría de la siguiente manera:

start_time	quantity_15	quantity_48	quantity_193
10:00 - 12/2/2018	2	1	0
11:00 - 15/2/2018	5	2	0
22:00 - 16/2/2018	1	2	1

Tabla 5: Ejemplo de *dataset* que contiene los intervalos.

Principalemente, el código usado para este paso hace uso de la función de *pandas* denominada `pivot()`:

```
# Prepare the new columns names for each station
df["quantity_index"] = "quantity_" + \
    df["from_station_id"].astype("str")

# We don't need from_station_id anymore
df = df.drop(columns=["from_station_id"])
```

```

# Make the pivot around quantity_index column and
# set the value of the column the same value as
# quantity from before saving quantity from
df = df.pivot(columns='quantity_index', values='quantity')

# If station any interval didn't have any trips for
# a station, then fill it with 0
return df.fillna(0)

```

Si se compara la tabla 3 con la tabla 6 se puede ver que la cantidad de datos ha sido reducida considerablemente y se ha seleccionado solo la información necesaria que la red necesitará.

3. Obtener la variables de tiempo: La columna *start_time* es una Serie con *timesteps* y por lo tanto tiene información más de la necesaria (año, mes, día, hora, minuto, segundo...). La única información temporal que se necesita son un subconjunto de dichos valores. En concreto, los valores que se han creído ser útiles son: la hora, el día de la semana (no es lo mismo un lunes que un sábado) y el mes (no es lo mismo un mes de invierno que de verano). Estos son los valores más simples que se han usado pero eso no quita que se puedan usar otros datos que aportan más información como por ejemplo: día del mes, año o el minuto.

Usando las variables de esta forma y no usando el *timestamp* directamente a la red, el modelo puede aprender patrones en función de variables más simples como la hora, el día de la semana o el mes, y no con un número de 64 bits que apenas aporta información alguna puesto que la red lo tomaría como valores ascendentes sin ningún tipo de criterio.

Para crear las nuevas columnas con los atributos que se han explicado anteriormente se han usado las siguientes líneas de código:

```

# Index and start_time column is the same

df['hour'] = df.index.hour           # Number between 0-23
df['day_of_week'] = df.index.dayofweek # Number between 0-6
df['month'] = df.index.month         # Number between 1-12

```

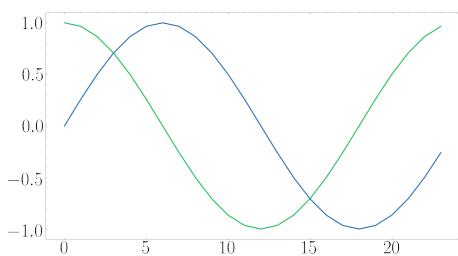
Al final de este paso existirán por tanto las siguientes columnas para cada uno de los intervalos: *start_time*, *hour*, *day_of_week*, *month* y una columna por cada estación. La columna *start_time* no se borra pues es la columna usada como índice y ayuda a la hora de entender la información con la que se trabaja. Pero no será usada por la red neuronal.

Usando el mismo ejemplo que los pasos anteriores, la tabla quedaría de la siguiente manera:

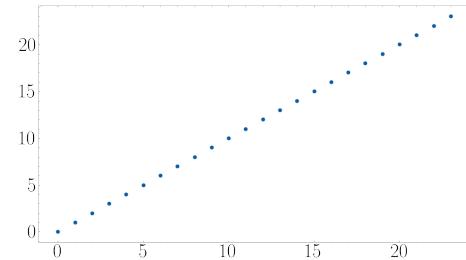
<i>start_time</i>	<i>quantity_15</i>	<i>quantity_48</i>	<i>quantity_193</i>	<i>hour</i>	<i>day_of_week</i>	<i>month</i>
10:00 - 12/2/2018	2	1	0	10	0	2
11:00 - 15/2/2018	5	2	0	11	3	2
22:00 - 16/2/2018	1	2	1	22	4	2

Tabla 6: Ejemplo de *dataset* final que será usada por la red neuronal.

En este punto también se estudió la posibilidad de guardar la información de las variables temporales en forma de señal con valores entre los rangos $[-1, 1]$ usando seno y coseno [52]. Esto permite que el modelo tenga acceso a la información temporal con valores continuos y no discretos como se puede ver en la figura 40 a continuación. Los resultados que se obtuvieron con ambas técnicas fueron similares, por lo que por simplicidad se dejó con valores discretos.



((a)) Hora con valores continuos de -1 a 1



((b)) Hora con valores discretos de 0 a 23

Figura 40: Posibles formas de representar la variable *hour*.

El DataFrame obtenido por este módulo se guarda en un fichero CSV llamado `intervals.csv` que será usado por los siguientes módulos cuyo contenido tendrá una estructura similar a la que se muestra en la tabla 6.

```
# Save df in a CSV file
df.to_csv("path/to/intervals.csv")
```

3.4. Generador de ventanas

El primer paso de este módulo es cargar el DataFrame generado en el anterior paso y dividirlo en tres partes: entrenamiento, validación y test. En concreto se han usado los datos del año 2017 para el entrenamiento, los datos del 2018 para la validación y los datos del 2019 para el test. A continuación se muestran las líneas de código usadas que realiza dicha división:

```
from datetime import datetime

def narrow_down_dataset(df, from_date, to_date):
    sub_df = df[(df.index >= from_date) & (df.index <= to_date)]

    # Just making sure intervals are sorted
    sub_df = sub_df.sort_values(by="start_time")

    return sub_df

'''

Splits the df into 3 parts depending on the year of the input
'''
def split_dataset(df, train_year=2017, validation_year=2018,
                  test_year=2019):
```

```

dfs = []

for year in [train_year, validation_year, test_year]:
    # 1st of Jan at 00:00
    from_date = datetime(year, 1, 1, 0)

    # 31st of Dec at 23:59
    to_date = datetime(year, 12, 31, 23, 59)

    dfs.append(narrow_down_dataset(df, from_date, to_date))

return dfs

```

df = pd.read_csv("/path/to/intervals.csv")
train_df, val_df, test_df = split_dataset(df)

Estas tres variables (train_df, val_df y test_df) serán usadas para las distintas fases que tiene el entrenamiento de una red neuronal como se ha explicado en la sección 2.5.1.

El módulo de generador de ventanas está basado en el código de un tutorial oficial de la librería de *Tensorflow* [53]. Básicamente este generador permite agrupar los intervalos de forma variable y permite ajustar al modelo a las necesidades que se necesiten. Los modelos de este trabajo harán un conjunto de predicciones basadas en una ventana de muestras consecutivas de los datos. Principalmente la ventana se puede ajustar con tres variables:

- Número de intervalos de entrada (*inputs*): Permite establecer la cantidad de intervalos que tendrá el modelo como valores de entrada. En este trabajo como se ha explicado en la anterior sección se está usando un intervalo de tamaño una hora. Por lo que si se quiere que el modelo tenga datos con un día previo para predecir, indicaremos al modelo que se quiere tener 24 intervalos previos. Hay que recalcar que es sumamente importante que los datos de la ventana estén ordenados y que sean intervalos contiguos.
- Número de intervalos de salida (*labels*): Permite establecer la cantidad de intervalos que se quieren predecir, es decir, la cantidad de intervalos que el modelo devolverá. En este trabajo por cada modelo se han probado distintas configuraciones cuyos resultados se mostrarán más adelante. Cabe destacar que cuánto mayor sea este valor y por lo tanto mayor cantidad de predicciones que el modelo realizará, el modelo se espera que sea menos preciso. Se denomina *labels* (etiquetas en español), puesto que son los intervalos que se van a usar para predecir las etiquetas de *quantity*.
- Desplazamiento (*offset*): Es una variable que permite establecer un espacio entre el primer intervalo del *input* y el primer intervalo de *labels*. Esto es útil porque se puede preparar modelos que dados 24 intervalos de entrada, se intente prever el mismo intervalo pero de un día futuro.

A continuación se muestran algunos ejemplos de ventanas con distintas configuraciones:

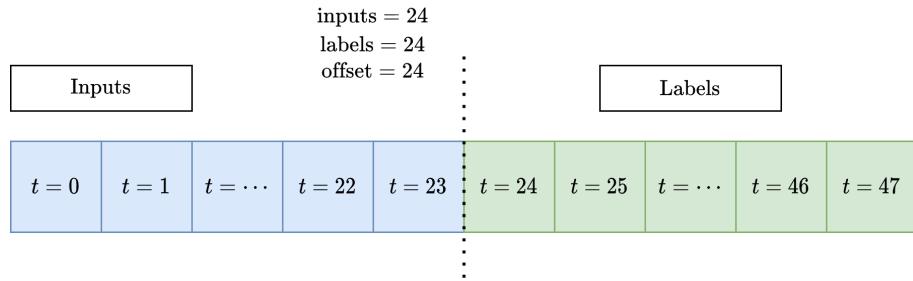


Figura 41: Ventana con un múltiples entradas y múltiples salidas

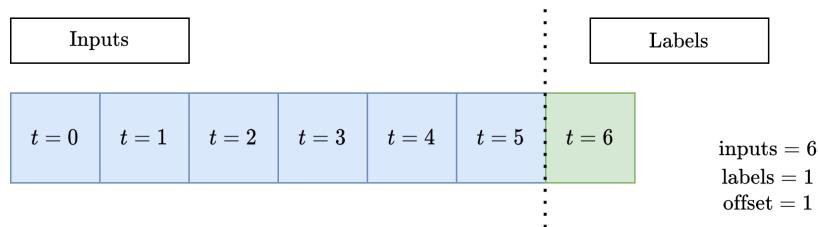


Figura 42: Ventana con un único intervalo de salida

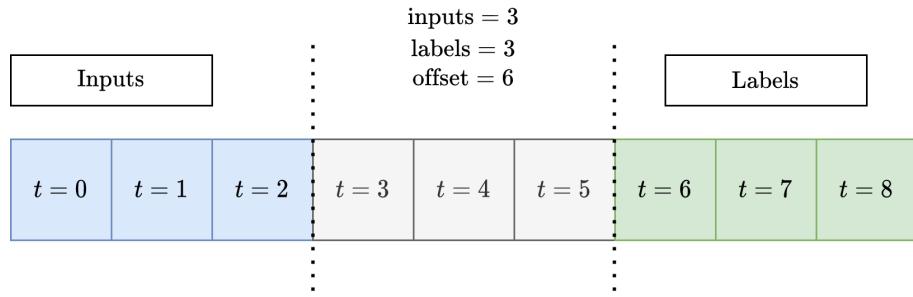


Figura 43: Ventana con *offset*

Cada caja representa un intervalos. El color azul representa que serán intervalos que el modelo usará como entrada, es decir, cada caja azul tiene asociado un vector con *hour*, *day_of_week*, *month* y la cantidad de viajes iniciados para todas las estaciones en dicho intervalo. Todos los vectores que representan a los intervalos de entrada, como se ha explicado en la sección 3.3.2, se transformará en un vector de una única dimensión. Las cajas verdes por otro lado representan los intervalos que las redes neuronales van a predecir. Por cada intervalo se creará un vector con tantos elementos como estaciones haya en el sistema siendo cada valor la predicción para cada estación.

Las predicciones por otro lado se pueden realizar de dos formas distintas:

- Predicción única: Dado un conjunto de datos de entrada se realizarán todas las predicciones a partir de esos datos en una sola iteración.

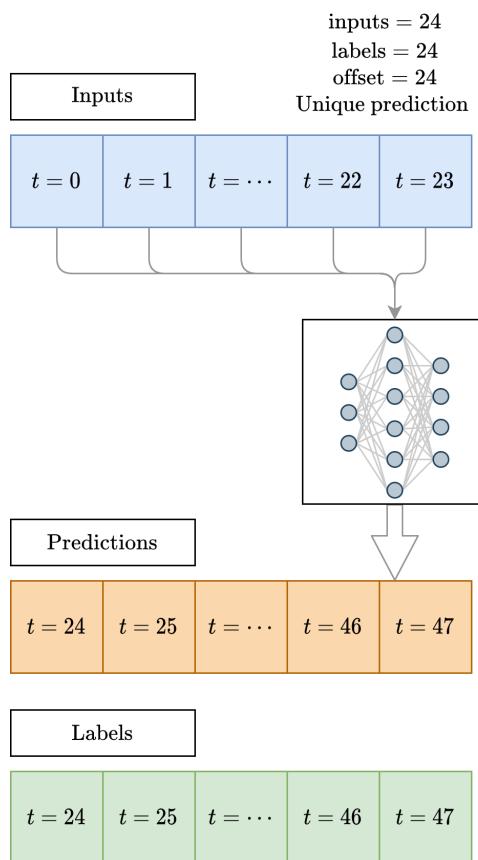


Figura 44: Predicciones únicas.

- **Predicción auto-regresiva :** Se realizará primero la predicción del intervalo más próximo en el futuro. Dicha predicción será usado junto con un vector de estados para poder calcular la siguiente predicción. Visto gráficamente:

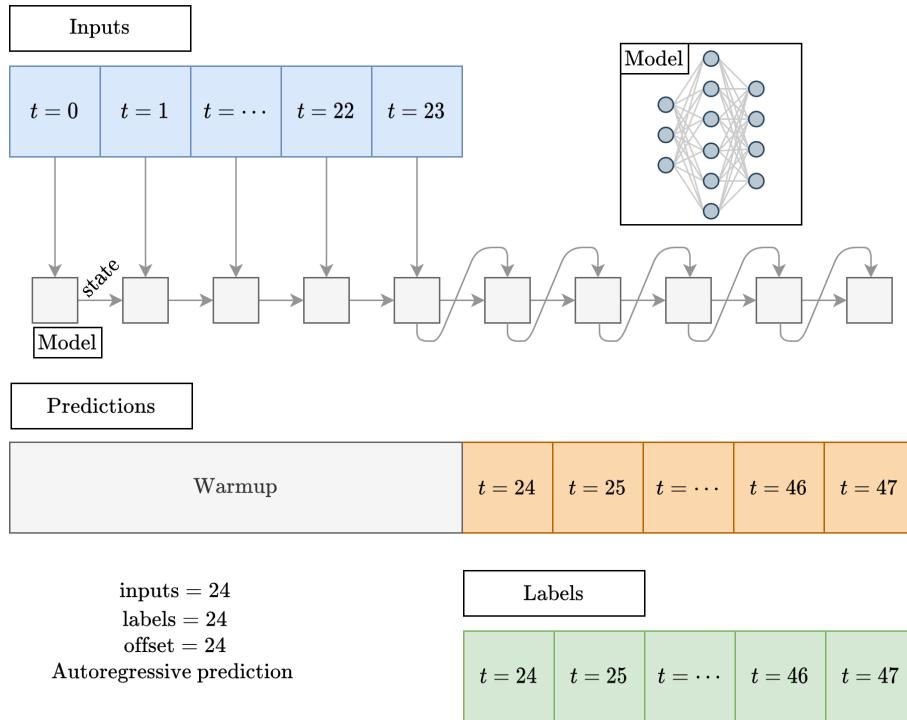


Figura 45: Predicción auto-regresiva.

Una vez que se tengan las ventanas, hay que hacer que la ventana se pueda deslizar y de ese modo generar un nuevos vectores que servirán para el entrenamiento. Pongamos por ejemplo, que se dispone de un *dataset* con 7 intervalos. Se quiere entrenar a un modelo que reciba 3 intervalos de entrada y 1 de salida. Por lo que el modelo se entrenará con un lote (*batch*) de tamaño 4. Visto gráficamente, el funcionamiento de una ventana deslizante:

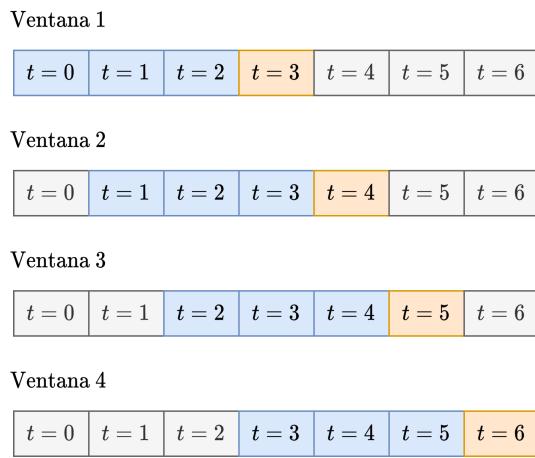


Figura 46: Funcionamiento de una ventana deslizante.

3.4.1. Código del generador de ventanas

Para este módulo se ha generado una clase que se encarga de crear objetos y se basa en gran parte en el tutorial oficial de *Tensorflow* [31] que ha sido modificado para suplir las necesidades del proyecto. El código usado se puede ver en el Apéndice D.

A continuación se van explican todas los métodos usados en la clase por partes.

- Índices y *offsets*: El método `__init__` (constructor en *Python*) incluye toda la lógica necesaria para los índices de entrada y de etiquetas. También toma los `DataFrames` de entrenamiento, validación y test como entrada. Estos serán convertidos a `tf.data.Datasets` de ventanas más tarde. Por último, recibe el índice de la columna donde se encuentra la información temporal. Este índice es un valor igual a -3 , pues como se puede ver en la tabla 6, las tres últimas columnas son las que contienen información sobre la hora, día de la semana y mes.
- División de la ventana entre *input* y *label*: Dada una lista de intervalos consecutivas, el método `split_window()` convertirá en una ventana de entradas y una ventana de etiquetas.
- Creación de los `tf.data.Datasets`: El método `make_dataset()` tomará un `DataFrame` de intervalos y lo convertirá en un `tf.data.Dataset` de pares (`input_window`, `label_window`). Se han usado *batches* de tamaño 32.
- La clase `WindowGenerator` contiene los `DataFrame` de entrenamiento, validación y test. Por lo que al final, se añaden las propiedades para acceder a ellos que son de tipo `tf.data.Datasets` que hacen uso del método anterior `make_dataset`:

4. MODELOS USADOS

En esta sección se explican los modelos que se han desarrollado. Se ha creado un modelo base y cuatro redes neuronales con cuatro arquitecturas distintas. Los resultados de estos modelos se pueden ver en la Sección 5.

4.1. Modelo base

Antes de definir las redes neuronales de este trabajo, se ha realizado un estudio de otros trabajos y resultados similares que se puede observar en la sección 2.7. Esto permite poder usar los resultados de otros proyectos como punto de referencia, sabiendo si el desarrollo de esta práctica está mejorando los resultados obtenidos por otros proyectos o no. Además de tener los puntos de referencias de otros proyectos, se ha definido un modelo base.

Un modelo base es un modelo trivial que se suele desarrollar cuando se quiere resolver un problema de predicción o clasificación para saber si el resto de modelos que se van a desarrollar mejorarán el resultado. En este proyecto, el modelo base desarrollado no tiene lógica y simplemente devuelve el valor que hubo con una semana de anterioridad como predicción.

A continuación se puede ver gráficamente las predicciones del modelo respecto a los valores reales:

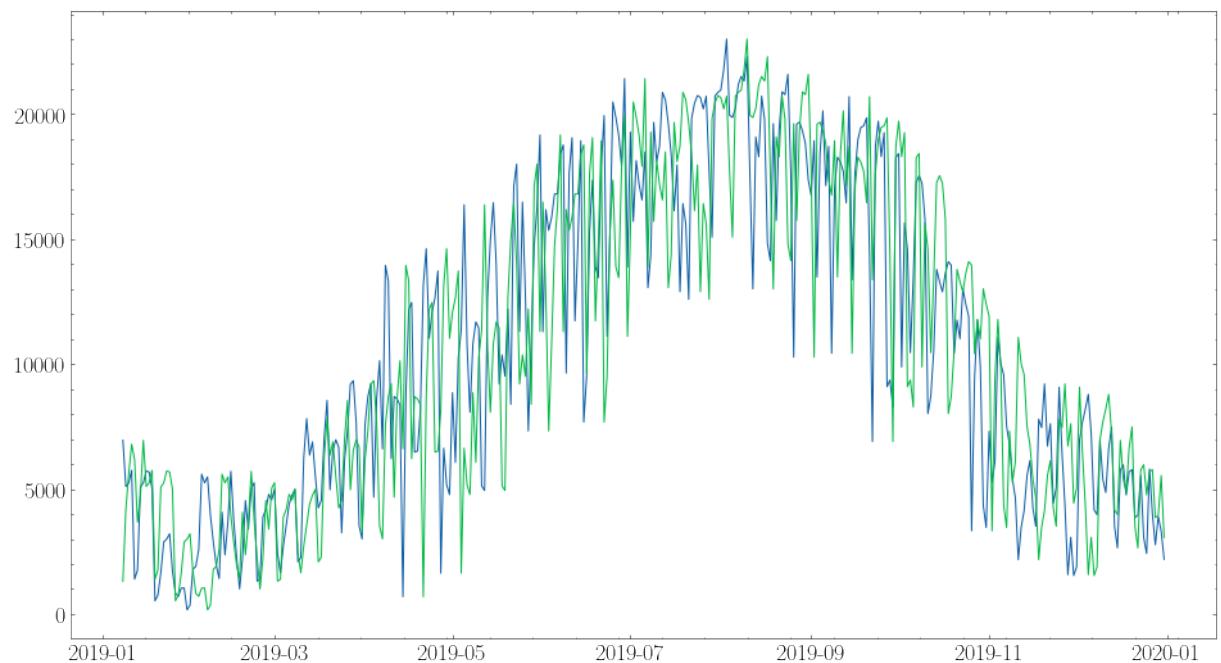


Figura 47: Predicciones del modelo básico

Se puede observar que las predicciones son los mismos valores reales pero desplazados una semana. Una vez desarrollado este modelo, se han calculado las métricas que se muestran en la sección 5 junto con el resto de los resultados de los otros modelos.

4.2. Modelos con redes neuronales

4.2.1. Modelo denso

Este modelo es un modelo con solo capas de tipo *forward-pass*. Tiene otras dos capas pero que no implementan lógica asociada a lo que es la red neuronal en sí, sino que son capas de traducción de matriz a vector y viceversa como se explicará a continuación. Gráficamente la red que se ha definido es la siguiente:

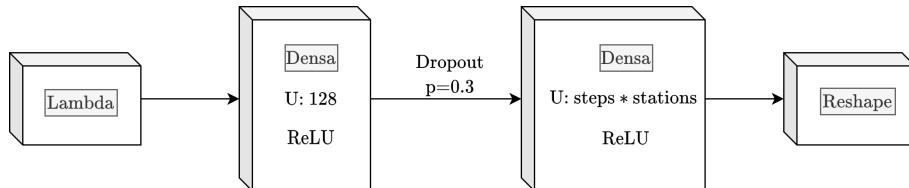


Figura 48: Modelo denso

La primera capa oculta tiene 128 unidades. La cantidad de capas y cantidad de neuronas que tiene una capa son parámetros que se han puesto de forma pseudoaleatoria. Se han realizado distintas combinaciones probando distintas cantidades de unidades y 2 capas densas con 128 unidades en la primera es la combinación que mejores resultados ha obtenido. La segunda capa oculta tiene la misma cantidad de unidades que valores tiene que devolver el modelo, que es un valor calculado por la multiplicación entre el número de estaciones con las que se está trabajando, 633, por el número de intervalos que se quieren calcular. Por ejemplo, si se quiere calcular un único intervalo para todas las estaciones, el vector de dicha capa será de 633 elementos, si se quiere calcular dos intervalos, el número de elementos del vector será $1266 = 2 \times 633$ y así sucesivamente. Esta capa es común a todos los modelos que se explicarán de aquí en adelante.

La última capa es una capa de tipo *Reshape*. Esta capa no tiene ni función de activación ni parámetros que ajustar porque lo único que hace es convertir el vector de la capa anterior a una matriz más comprensible y con la que se puede trabajar de forma más cómoda con el *dataset* donde cada fila será un intervalo y cada columna será una estación. Los valores de esta matriz serán por tanto la predicción que el modelo devuelve para el intervalo y estación definidas por su fila y columna en el que se encuentran.

Siguiendo la misma lógica, la primera capa es de tipo *Lambda* y se encarga de realizar la tarea opuesta, dada una matriz obtener un vector. Las capas *Lambda* son capas en las que el usuario define una función que se quiera ejecutar.

El código de este modelo se ha definido como se muestra a continuación:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape, Dense, Dropout, Lambda

# `steps` is a variable which has the number of intervals to be predict
steps = 0

# `stations` is the number of stations in the bike network
stations = 0

dense_model = Sequential([
    Lambda(lambda x: x),
    Dense(128, activation='relu'),
    Dropout(0.3),
    Dense(steps * stations, activation='relu'),
    Reshape((steps, stations))
])
```

```

# Matrix to vector
Lambda(lambda x: x[:, -1:, :]),

# Input layer
Dense(128, activation="relu"),

Dropout(0.3),

# Output layer
Dense(steps * stations),

# Vector to matrix
Reshape([steps, stations])
])

```

Los resultados del modelo se pueden ver junto al resto de resultados en la sección 5.

4.2.2. Modelo recurrente básico

Este modelo es el mismo modelo que el denso pero añadiendo una nueva capa recurrente simple explicada en la sección 2.6.1. La capa recurrente simple es una capa que nos permite usar información de otros intervalos anteriores que serán usados para la predicción final. Esta capa recurrente tiene 100 unidades y gráficamente el modelo se puede representar de la siguiente forma:

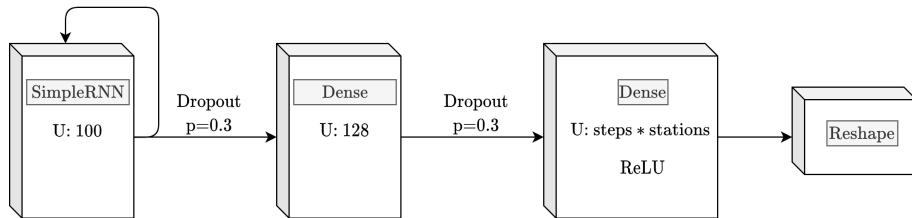


Figura 49: Modelo recurrente básico

Como se puede observar es una extensión del modelo denso explicado en la sección anterior pero añadiendo una nueva capa de tipo *SimpleRNN*. El código de este modelo se ha definido como se muestra a continuación:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape, Dense, Dropout, Lambda, SimpleRNN

# `steps` is a variable which has the number of intervals to be predict
steps = 0

# `stations` is the number of stations in the bike network
stations = 0

rnn_model = Sequential([
    # RNN layer
    SimpleRNN(100, return_sequences=True),

    Dropout(0.3),
    Dense(128),
    Dropout(0.3),

    # Output layer
    Dense(steps * stations),

    # Vector to matrix
    Reshape([steps, stations])
])

```

Los resultados del modelo se pueden ver junto al resto de resultados en la sección 5.

4.2.3. Modelo recurrente LSTM

Este modelo es igual que el modelo recurrente básico pero sustituyendo la capa recurrente por una capa LSTM la cual es más compleja y permite que la red aprenda pueda usar mayor cantidad de información del pasado como se explicó en la sección 2.6.6. Esta capa recurrente tiene 128 unidades y gráficamente el modelo se puede representar de la siguiente forma:

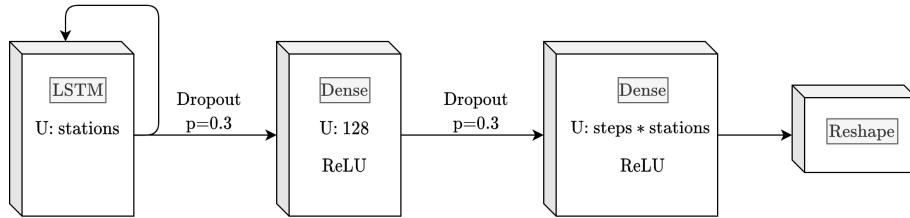


Figura 50: Modelo recurrente LSTM

Como se puede observar es el mismo modelo pero sustituyendo la capa *SimpleRNN* por una LSTM. El código de este modelo se ha definido como se muestra a continuación:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Reshape, Dense, Dropout, Lambda, LSTM

# `steps` is a variable which has the number of intervals to be predict
steps = 0

# `stations` is the number of stations in the bike network
stations = 0

lstm_model = Sequential([
    # LSTM layer
    LSTM(stations, return_sequences=False),

    Dropout(0.3),
    Dense(128, activation="relu"),

    # Output layer
    Dropout(0.3),
    Dense(steps * stations, activation="relu"),

    # Vector to matrix
    Reshape([steps, stations])
])
  
```

Los resultados del modelo se pueden ver junto al resto de resultados en la sección 5.

4.2.4. Modelo Autoregresivo (AR)

Un modelo AR es otro tipo de redes neuronales recurrentes. Esta red tiene como peculiaridad que el resultado obtenido se usa como valor de entrada para el cálculo de la siguiente iteración. El modelo generado gráficamente es el siguiente:

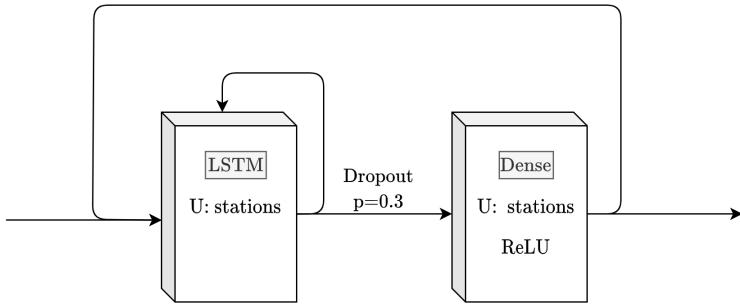


Figura 51: Modelo recurrente AR

El modelo tiene una capa LSTM junto con una capa densa. La salida de esta capa que es la predicción del modelo se reintroduce en el modelo para predecir el resto de predicciones.

Tensorflow no dispone de un modelo en su librería herramientas que supliese las necesidades que se buscaban y por esa razón, se ha tenido que implementar este modelo usando otra técnica. Se ha creado una clase nueva que hereda de `tf.keras.Model` y la cual tiene tres métodos: `__init__`, `warmup` y `call`. De esta forma se pueden generar modelos que puedan tener arquitecturas singulares y distintas a las *pass-forward*. Para la implementación se ha seguido la documentación de *Tensorflow* [31] y *Keras* [46]. El código desarrollado para esta clase se puede leer en el Apéndice E.

Para crear un nuevo modelo AR simplemente se genera una nueva instancia de la siguiente manera:

```
AutoRegressive(lstm_units=stations, steps=steps, stations=stations)
```

Los resultados del modelo se pueden ver junto al resto de resultados en la sección 5.

4.3. Ventanas

Como se ha explicado en la sección 3.4, se pueden generar predicciones en función de los argumentos de entrada. Es decir, se puede crear un modelo que dado una cantidad X de intervalos, pueda predecir una cantidad Y de intervalos. Se quería comparar diferentes ajustes que se pueden realizar a las ventanas y ver los resultados obtenidos. En definitiva, se ha creado un modelo distinto para cada uno de los tamaños de ventana.

De aquí en adelante, se hará referencia a las configuración de ventana con tuplas, donde el primer término es el tamaño de intervalos de entrada y el segundo elemento el tamaño de intervalos que el modelo predice. Por ejemplo dada la siguiente tupla 12-5, quiere decir que el modelo desarrollado usa una ventana con 12 intervalos de entrada y 5 intervalos de salida.

En total se han usado 15 configuraciones de ventana distintas: 3-1, 5-1, 8-1, 8-3, 8-5, 12-1, 12-3, 12-5, 24-1, 24-3, 24-5, 36-8, 36-12, 48-12, 48-24. Como se han desarrollado 4 arquitecturas de neuronales diferentes, al final se han desarrollado 60 modelos más el modelo básico.

4.4. Entrenamiento, validación y test de las redes neuronales

Una vez creado los modelos, se ha codificado una función para automatizar el proceso de entrenamiento, validación y test de cada uno de los modelos, además de obtener las métricas que se necesitarán para poder estudiar y comparar los distintos modelos. El código se puede leer en el Apéndice F.

El código se divide en tres funciones distintas:

- La función `compile_and_fit()`, dado un modelo de arquitectura de red (denso, RNN, LSTM o AR), compilará dicho modelo usando como función de coste la pérdida de Huber y como optimizador para el descenso del gradiente, el optimizador de *Adam*. La tasa de aprendizaje usada ha sido un valor entre 0.001 y 0.0001 dependiendo del modelo. Para obtener este valor, se ha hecho uso de la técnica explicada en la Sección 2.5.7 en la cual se calcula la tasa de aprendizaje de forma iterativa y estudiando que valores hacen que el descenso del gradiente converja de forma más rápida y constante.

El optimizador usado ha sido el optimizador de *Adam*. Esta decisión se ha tomada con el mismo proceso que la decisión de cuantas capas y cuantas neuronas usar en la red, a base de prueba y error. Seguramente, que exista una combinación que obtenga resultados mejores, pero desde la experiencia de los resultados obtenidos con las distintas comparaciones, los modelos no suelen mejorar en exceso por este tipo de decisiones, es mucho más importante otros hiperparámetros como la función de activación a usar, una tasa de aprendizaje óptima o el uso de *Dropout* entre capas.

Otra técnica para evitar la aparición de *Overfitting* es el uso de una función denominada `EarlyStopping` que parará el proceso de aprendizaje si considera que el modelo está empezando a memorizar y no aprender o si después de una `patience` igual a 10 `epochs` el modelo no ha aprendido de forma considerable respecto al pasado. Esto se hace a partir de `callbacks`, que son funciones que se ejecutan tras ejecutar un `epoch`. Otro `callback` usado ha sido una función que en tiempo real dibuja las gráficas de como está yendo el proceso de aprendizaje con las métricas con el `dataset` de entrenamiento como el `dataset` de validación.

En cuanto a la función de coste usada, se explicará en la siguiente sección junto con las métricas usadas para medir el desempeño de los modelos.

- La función `compile_and_fit()` solo trabaja con un modelo y con un tamaño de ventana en concreto. Por otro lado, se tiene la función `model_generator()` que dado un modelo, se encarga de crear todas ventanas con los distintos tamaños y entrenar el modelo para cada una de las ventanas. Como se ha explicado previamente, en total se han usado 15 tamaños de ventana distinto, por lo que una llamada a la función `model_generator()` provoca que se entrenen 15 modelos distintos y por lo tanto el tiempo de este proceso suele durar entre 2 y 5 horas dependiendo del modelo siendo entrenado.
- Por cada modelo entrenado y para no desperdiciar los resultados, la función `get_metrics()`

se encarga de computar dichos datos y guardarlos en un CSV que será utilizado por otras funciones más adelante para poder mostrar los resultados y dibujar distintas gráficas.

5. RESULTADOS

En esta sección se mostrarán todos los resultados obtenidos con los diferentes modelos. En total se han desarrollado 4 modelos (1 modelo básico y 3 redes neuronales). Estos modelos se han ido iterando desde un modelo básico hasta modelos más complejos. Dicho proceso de mejora se ve reflejado en las métricas obtenidas.

5.1. Métricas usadas

Se han usado 6 métricas distintas para medir el rendimiento de los modelos. Además, una de esas métricas, la pérdida de Huber (ver sección 2.5.2) se ha usado como función de coste.

5.1.1. Pérdida de Huber

Como se explica en la Sección 2.5.2, esta función de pérdida no prioriza los datos atípicos sino que trata a todos los datos de forma similar. Esto es muy útil, para el problema que queremos resolver.

En el dataset que se está usando, un dato que no siga el patrón (*outlier*) no significa que sea un error que se deba de tener en cuenta como puede ocurrir en otros tipos de problemas. Por ejemplo, en una estación de bicicletas justo en frente de una facultad recibe una gran demanda un jueves a las 17 mientras que el resto de jueves no ha tenido tanta demanda para dicho intervalo. Esto se debe a un acontecimiento único, por ejemplo un examen en dicha facultad para ese día. Este dato, al ser un dato válido y no un error como tal, la red no aprenderá de inmediato que los jueves habrá dicha demanda sino que harán faltas múltiples días en los que se repita para que la red comience a ajustarse a dichos datos y por eso la pérdida de Huber se adapta bien a este tipo de problema. La ecuación de la pérdida de Huber es la siguiente:

$$\text{Huber} = \begin{cases} MSE & \text{si } |\hat{y} - y| \leq \delta, \\ MAE & \text{e.o.c.} \end{cases} \quad (95)$$

Para la implementación del código se ha usado la propia implementación de la librería de *Keras*:

```
from tf.losses import Huber
```

5.1.2. MAE

El error absoluto medio (MAE) es una de las métricas más básicas y rápidas de calcular. Se calcula usando la media entre las diferencias entre los valores reales y los valores predichos como se puede ver a continuación:

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (96)$$

Para hacer uso de esta ecuación en *Keras*, se puede importar de la siguiente forma:

```
from tf.metrics import MeanAbsoluteError
```

5.1.3. MSE y RMSE

Estas dos métricas indican básicamente lo mismo. El MSE es un valor que se obtiene al calcular la media entre todos los valores obtenidos al aplicar el cuadrado a la diferencia entre el valor real y el valor predicho como se puede ver a continuación:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2 \quad (97)$$

Por otro lado, RMSE es la raíz cuadrada del RMSE como se puede observar a continuación:

$$\text{RMSE} = \sqrt{\text{MSE}} \quad (98)$$

Para poder hacer cálculo de MSE y RMSE se pueden importar desde *tensorflow*.

```
from tf.losses import MeanSquaredError
from tf.metrics import RootMeanSquaredError
```

5.1.4. MSLE y RMSLE

Estas dos métricas se usa porque hay multitud de proyectos, entre ellos, la competición de *Kaggle* (ver sección 2.7.3) que las usan y de esa forma poder comparar los resultados. La explicación del MSLE se puede ver en la sección 2.5.2. RMSLE es igual que MSLE pero aplicándole una raíz cuadrada:

$$\begin{aligned} \text{MSLE} &= \frac{1}{n} \sum_{i=1}^n \left(\log \left(\frac{y_i + 1}{\hat{y}_i + 1} \right) \right)^2 \\ \text{RMSLE} &= \sqrt{\text{MSLE}} \end{aligned} \quad (99)$$

Para poder hacer cálculo de MSLE se puede importar la función de *Keras*. Esta función además nos sirve para poder definir la función RMSLE como se muestra a continuación:

```
from tf.keras.losses import MeanSquaredLogarithmicError
def RootMeanSquaredLogarithmicError(y_true, y_pred):
    msle = MeanSquaredLogarithmicError(y_true, y_pred)
    return tf.keras.backend.sqrt(msle)
```

5.2. Resultados del modelo básico

El modelo básico como se ha explicado en la sección 4.1, devuelve la cantidad de viajes que se iniciaron para un intervalo la anterior semana a la cual se quiere predecir el dato. Calculando todas las métricas, los resultados de este modelo son los siguientes:

Métricas	val	test
MSLE	0.221549	0.239364
MSE	3.383871	3.699979
MAE	0.603720	0.647458
HUBER	0.462739	0.499985
RMSE	1.913998	1.914635
RMSLE	0.470690	0.489248

Tabla 7: Métricas obtenidas del modelo básico.

Estos valores son los que se usarán como referencia para poder comparar los modelos que se desarrollen y comprobar que se está consiguiendo una mejora.

5.3. Redes neuronales

A continuación se muestran los resultados de todos los modelos con las distintas ventanas que se han entrenado para estudiar como se comporta. Todos los modelos tienen los mismo hiperparámetros que se han explicado en la sección 2.2.1. A continuación, se puede ver una tabla con todos los resultados obtenidos. Las columnas representan los modelos usados, los cuales todos tienen las métricas obtenidas tanto para el *dataset de validation* como el *dataset de testing*. Las filas están divididas tanto por las métricas usadas como por el tamaño de ventana para cada modelo. En total se han desarrollado 60 modelos (15 tamaños de ventanas distintos por 4 modelos de redes neuronales):

Windows	Dense		RNN		LSTM		AR		
	val	test	val	test	val	test	val	test	
(3, 1)	0.344	0.380	0.334	0.374	0.474	0.508	0.290	0.321	
(5, 1)	0.375	0.411	0.347	0.388	0.476	0.510	0.296	0.330	
(8, 1)	0.410	0.445	0.352	0.389	0.477	0.512	0.291	0.324	
(8, 3)	0.348	0.380	0.331	0.369	0.293	0.329	0.295	0.331	
(8, 5)	0.373	0.411	0.346	0.385	0.301	0.342	0.303	0.342	
(12, 1)	0.435	0.472	0.378	0.418	0.472	0.506	0.288	0.320	
(12, 3)	0.344	0.379	0.334	0.370	0.294	0.330	0.302	0.338	
HUBER	(12, 5)	0.371	0.409	0.346	0.386	0.303	0.340	0.307	0.344
	(24, 1)	0.460	0.495	0.434	0.474	0.472	0.507	0.290	0.323
	(24, 3)	0.353	0.386	0.332	0.370	0.295	0.331	0.300	0.335
	(24, 5)	0.363	0.401	0.347	0.386	0.301	0.341	0.311	0.348
	(36, 8)	0.355	0.399	0.358	0.397	0.299	0.338	0.317	0.358
	(36, 12)	0.362	0.403	0.373	0.409	0.306	0.348	0.335	0.385
	(48, 12)	0.367	0.408	0.370	0.409	0.306	0.347	0.327	0.369
	(48, 24)	0.374	0.414	0.385	0.423	0.311	0.353	0.337	0.374

Windows	Dense		RNN		LSTM		AR	
	val	test	val	test	val	test	val	test
MSLE	(3, 1)	0.136	0.154	0.130	0.149	0.235	0.253	0.115 0.131
	(5, 1)	0.153	0.171	0.140	0.160	0.240	0.258	0.117 0.134
	(8, 1)	0.177	0.195	0.143	0.162	0.242	0.261	0.115 0.131
	(8, 3)	0.134	0.151	0.129	0.148	0.114	0.131	0.118 0.135
	(8, 5)	0.149	0.170	0.136	0.156	0.117	0.135	0.123 0.142
	(12, 1)	0.198	0.217	0.157	0.177	0.233	0.251	0.117 0.132
	(12, 3)	0.132	0.150	0.129	0.147	0.114	0.131	0.123 0.140
	(12, 5)	0.148	0.169	0.136	0.157	0.118	0.135	0.124 0.142
	(24, 1)	0.217	0.235	0.194	0.217	0.234	0.252	0.118 0.134
	(24, 3)	0.137	0.154	0.130	0.149	0.115	0.132	0.122 0.139
	(24, 5)	0.144	0.164	0.138	0.157	0.117	0.136	0.129 0.148
	(36, 8)	0.144	0.167	0.142	0.163	0.119	0.138	0.128 0.148
	(36, 12)	0.148	0.169	0.148	0.166	0.121	0.141	0.138 0.162
	(48, 12)	0.150	0.172	0.150	0.170	0.121	0.141	0.135 0.156
	(48, 24)	0.154	0.175	0.155	0.175	0.122	0.142	0.139 0.158

Windows	Dense		RNN		LSTM		AR	
	val	test	val	test	val	test	val	test
MSE	(3, 1)	2.518	2.802	2.513	2.831	4.958	5.302	1.623 1.793
	(5, 1)	3.228	3.526	2.678	3.022	5.017	5.375	1.684 1.899
	(8, 1)	3.885	4.144	2.755	3.018	5.028	5.392	1.622 1.826
	(8, 3)	2.792	3.012	2.454	2.736	1.740	2.025	1.692 1.976
	(8, 5)	3.185	3.550	2.730	3.031	1.890	2.284	1.839 2.155
	(12, 1)	4.278	4.593	3.237	3.514	5.017	5.358	1.582 1.785
	(12, 3)	2.746	3.017	2.557	2.830	1.784	2.081	1.810 2.052
	(12, 5)	3.171	3.510	2.692	2.998	1.929	2.240	1.895 2.187
	(24, 1)	4.653	4.974	4.393	4.749	5.032	5.389	1.604 1.818
	(24, 3)	2.831	3.069	2.448	2.724	1.751	2.038	1.749 2.002
	(24, 5)	2.990	3.330	2.701	2.980	1.872	2.222	2.122 2.403
	(36, 8)	2.724	3.146	2.928	3.220	1.743	2.111	2.014 2.404
	(36, 12)	2.880	3.225	3.210	3.475	1.870	2.272	2.327 2.842
	(48, 12)	2.965	3.333	3.098	3.372	1.855	2.246	2.093 2.476
	(48, 24)	3.081	3.431	3.377	3.653	1.950	2.350	2.357 2.649

Windows	Dense		RNN		LSTM		AR		
	val	test	val	test	val	test	val	test	
(3, 1)	0.508	0.550	0.519	0.565	0.699	0.738	0.468	0.504	
(5, 1)	0.546	0.587	0.543	0.592	0.674	0.713	0.475	0.515	
(8, 1)	0.594	0.636	0.542	0.586	0.672	0.711	0.469	0.508	
(8, 3)	0.524	0.563	0.521	0.565	0.469	0.512	0.455	0.498	
(8, 5)	0.546	0.591	0.535	0.581	0.477	0.524	0.461	0.506	
(12, 1)	0.642	0.685	0.561	0.609	0.702	0.741	0.444	0.481	
(12, 3)	0.519	0.560	0.520	0.562	0.469	0.512	0.458	0.501	
MAE	(12, 5)	0.545	0.590	0.537	0.583	0.478	0.522	0.463	0.506
	(24, 1)	0.677	0.718	0.629	0.674	0.704	0.743	0.446	0.485
	(24, 3)	0.526	0.564	0.522	0.567	0.473	0.515	0.453	0.494
	(24, 5)	0.536	0.580	0.539	0.585	0.477	0.523	0.467	0.511
	(36, 8)	0.548	0.599	0.551	0.598	0.477	0.523	0.475	0.523
	(36, 12)	0.556	0.604	0.562	0.606	0.485	0.534	0.488	0.545
	(48, 12)	0.559	0.608	0.571	0.617	0.485	0.533	0.479	0.530
	(48, 24)	0.570	0.618	0.582	0.626	0.491	0.539	0.490	0.534

Windows	Dense		RNN		LSTM		AR		
	val	test	val	test	val	test	val	test	
(3, 1)	1.587	1.674	1.585	1.683	2.225	2.303	1.273	1.339	
(5, 1)	1.797	1.877	1.637	1.739	2.241	2.319	1.297	1.378	
(8, 1)	1.972	2.037	1.660	1.738	2.244	2.322	1.274	1.351	
(8, 3)	1.671	1.735	1.567	1.655	1.318	1.424	1.301	1.406	
(8, 5)	1.785	1.885	1.653	1.740	1.375	1.511	1.356	1.468	
(12, 1)	2.070	2.143	1.796	1.876	2.238	2.317	1.258	1.337	
(12, 3)	1.659	1.738	1.598	1.679	1.336	1.441	1.346	1.434	
RMSE	(12, 5)	1.782	1.876	1.642	1.732	1.389	1.497	1.377	1.478
	(24, 1)	2.157	2.230	2.096	2.179	2.243	2.321	1.266	1.348
	(24, 3)	1.682	1.752	1.565	1.651	1.323	1.428	1.323	1.415
	(24, 5)	1.729	1.825	1.643	1.726	1.368	1.491	1.457	1.550
	(36, 8)	1.649	1.772	1.712	1.795	1.320	1.453	1.419	1.550
	(36, 12)	1.697	1.798	1.792	1.865	1.366	1.508	1.526	1.685
	(48, 12)	1.722	1.826	1.760	1.836	1.362	1.499	1.447	1.574
	(48, 24)	1.755	1.853	1.837	1.911	1.397	1.533	1.535	1.627

Windows	Dense		RNN		LSTM		AR		
	val	test	val	test	val	test	val	test	
RMSLE	(3, 1)	0.367	0.391	0.360	0.385	0.484	0.501	0.339	0.360
	(5, 1)	0.390	0.413	0.373	0.399	0.488	0.506	0.342	0.365
	(8, 1)	0.420	0.440	0.377	0.401	0.490	0.509	0.339	0.361
	(8, 3)	0.365	0.388	0.359	0.384	0.337	0.362	0.343	0.367
	(8, 5)	0.385	0.411	0.368	0.394	0.341	0.367	0.350	0.376
	(12, 1)	0.444	0.465	0.395	0.420	0.481	0.500	0.341	0.363
	(12, 3)	0.362	0.387	0.358	0.383	0.337	0.361	0.350	0.374
	(12, 5)	0.384	0.410	0.369	0.395	0.342	0.367	0.351	0.376
	(24, 1)	0.464	0.484	0.439	0.464	0.482	0.501	0.343	0.366
	(24, 3)	0.369	0.392	0.360	0.385	0.339	0.363	0.349	0.372
	(24, 5)	0.378	0.404	0.370	0.396	0.342	0.368	0.359	0.384
	(36, 8)	0.379	0.408	0.377	0.403	0.344	0.370	0.357	0.385
	(36, 12)	0.384	0.411	0.384	0.407	0.347	0.375	0.371	0.402
	(48, 12)	0.387	0.414	0.387	0.412	0.347	0.375	0.367	0.395
	(48, 24)	0.392	0.418	0.394	0.418	0.350	0.377	0.372	0.397

También se han desarrollado gráficas para poder visualizar las métricas más fácilmente usando el dataset de *testing*. En todas las gráficas hay una línea horizontal que representa la métrica obtenida por el modelo base:

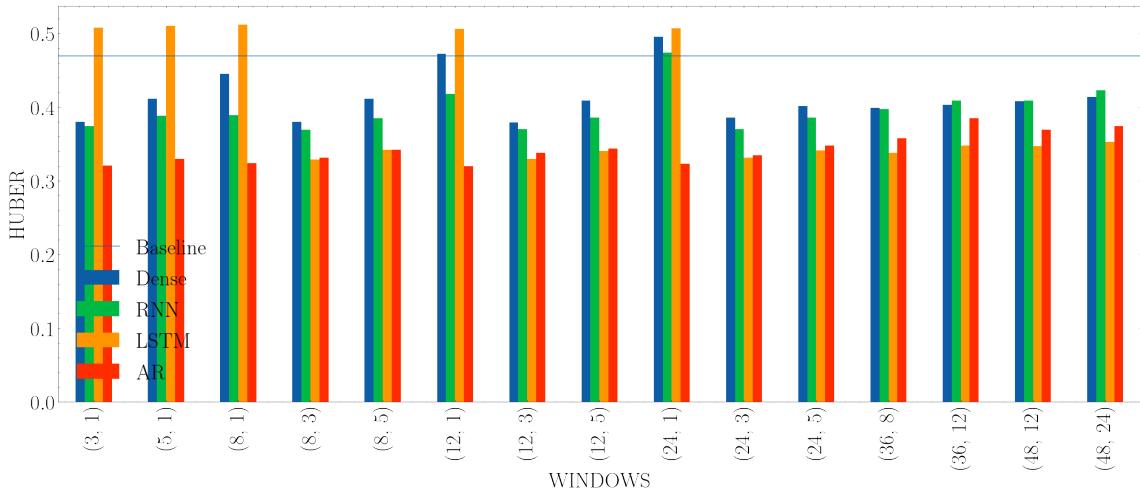


Figura 52: Resultados usando pérdida de Huber

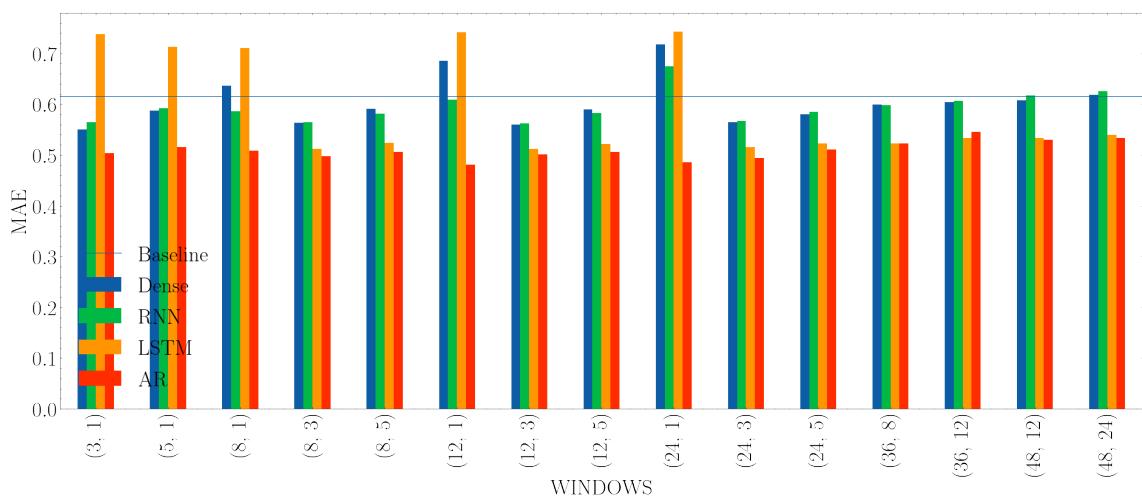


Figura 53: Métricas usando MAE

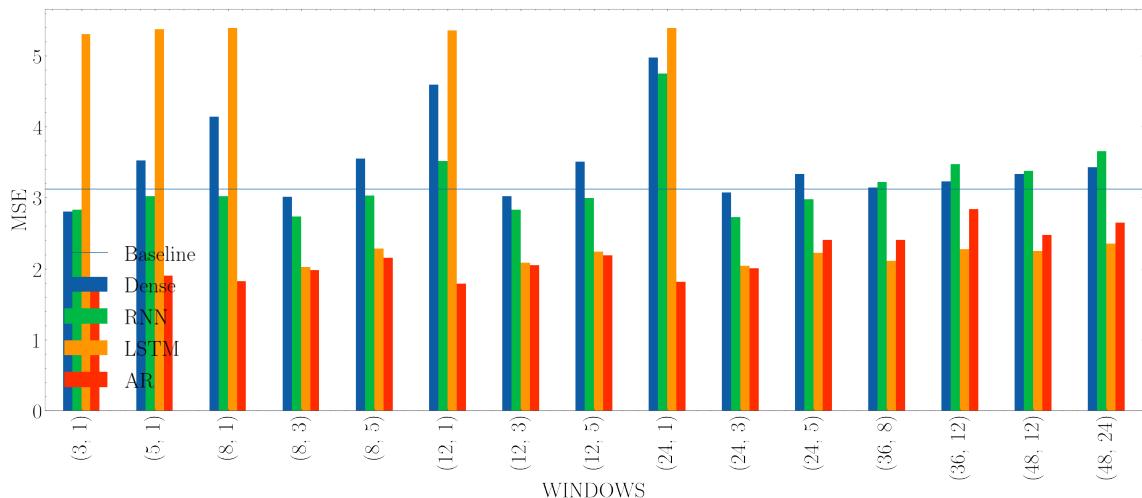


Figura 54: Métricas usando MSE

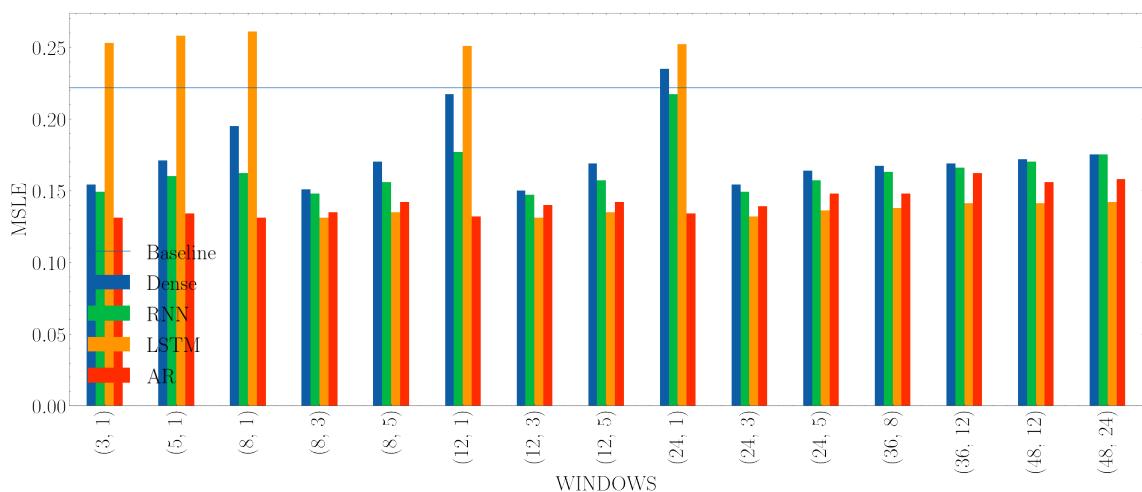


Figura 55: Métricas usando MSLE

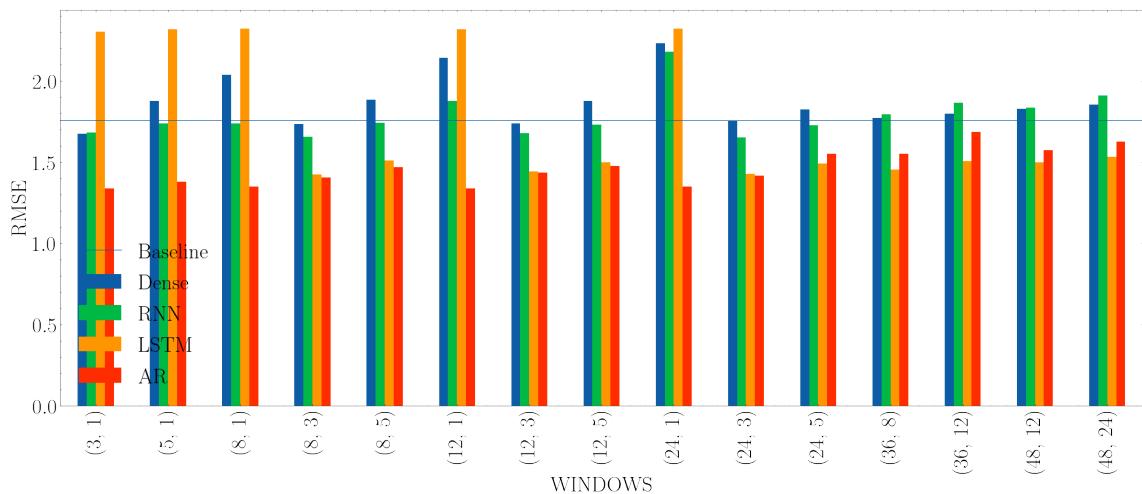


Figura 56: Métricas usando RMSE

6. PROBLEMAS ENCONTRADOS

Durante el desarrollo de este trabajo nos hemos encontrado con distintos problemas tanto a la hora de desarrollar las redes neuronales como al preprocesado necesario para poder crear el *dataset*.

6.1. Estructura del dataset

Toda red neuronal tiene como requisito tener una arquitectura bien definida antes de poder ser creada. Entre otros parámetros, hay que definir tanto la entrada como la salida con la que el modelo trabajará. Al principio del desarrollo, al entrenar las primeras redes de prueba, obteníamos valores muy malos para lo que estábamos esperando. Todo ello era consecuencia de la mala definición que se había llevado a cabo para entrenar las redes.

Principalmente el error se encontraba en qué estaba contemplando desde el principio que una red pudiese aceptar una matriz como entrada, lo cual, no es cierta dicha afirmación. Una red siempre tiene que trabajar con vectores para las variables x e y .

Posteriormente, desarrollé una estructura que tenía más sentido y con la cual la red obtenía mejores resultados. El problema de esta solución fue el tiempo de entrenamiento cuando se intentaba entrenar a una red básica. Era inviable seguir con esta estructura de datos.

Finalmente, tras varias reuniones con el tutor y con ayuda de un estudiante de doctorado llegamos a la conclusión que la mejor forma de entrenar a la red neuronal era con la estructura de datos que se ha explicado con anterioridad en este trabajo. Esta solución a pesar de ser muy intuitiva y fácil de entender, me ha costado una gran cantidad de tiempo llegar a ella.

6.2. Desarrollo del preprocesado

Para poder desarrollar una estructura de datos con que se puedan generar los modelos de las redes neuronales, es necesario previamente reestructurar el *dataset* que usamos a un *dataset* con el que se puedan generar las ventanas necesarias.

Para ello es necesario usar algún tipo de librería que facilite este proceso. En este trabajo se ha decidido usar `DataFrame` de *pandas* junto con otras librerías como *numpy* o *tensorflow*, las cuales eran mi primera vez usándolas pero que me he podido adaptar con facilidad.

A pesar de ello, me he encontrado con más problemas de los esperados a la hora de usar estas librerías y los cuales me han ralentizado el desarrollo. Además, muchas de las funciones que han sido desarrolladas no estaba seguro de si realizaban la tarea que se les encomendaba y por ello, he dedicado también parte de tiempo a escribir algunos test unitarios para asegúrame que el código era bueno.

Muchos de estos problemas tienen su origen a la hora de trabajar con más de 3 dimensiones. A partir de ese punto, es mucho más complejo para el cerebro humano poder tratar estructuras de este estilo y es importante realizar un estudio y un pseudocódigo de lo que se quiere hacer

previamente. Esto sin duda alguna trataré de aplicarlo a proyectos futuros: dedicar más tiempo al diseño.

6.3. Problema de *overfitting*

El problema del *overfitting* es un problema común cuando se entrena red neuronales. Como se explica en la sección 2.5.7, el *overfitting* aparece tras un período de tiempo en el que la red ya ha aprendido y comienza a memorizar los datos, provocando que no se adapte a nuevas situaciones correctamente y pudiendo incluso empeorar los resultados del modelo.

Por esa razón, he dedicado parte de mi trabajo al estudio de como evitar el *overfitting*. En principio se usó la estrategia de trabajar con reguladores, que son ecuaciones con las cuales los pesos de una red se cambian con más lentitud. Pero esta solución, no ha tenido los resultados esperados. Lo que se conseguía con ello, era que el *overfitting* apareciese pero en iteraciones del entrenamiento mucho más avanzadas, por lo que desestimé este el uso de reguladores.

Finalmente, en los modelos actuales se han usado dos técnicas para evitar el *overfitting*. En primer lugar, se ha usado una técnica por la cual se elige una tasa de aprendizaje con la cual el modelo converja a la hora del cálculo del gradiente. Esta técnica se basa en usar varios valores de la tasa de aprendizaje y estudiando como el descenso del gradiente se comporta respecto a ello. En segundo lugar, se ha usado la técnica del uso de *Dropout* (ver sección 2.5.7), que básicamente activa y desactiva conexiones entre neuronas en la red de forma aleatoria para que ninguna sea prescindible, es decir, que contenga gran parte de conocimiento respecto a los patrones que se quieren predecir.

7. CONCLUSIONES

7.1. Estudio de los resultados

Las gráficas presentadas en la anterior sección, muestran las métricas obtenidas usando el DataFrame de test. Usando estos datos, refleja como de bueno son los modelos ante datos que nunca antes había visto y como se comportaría en un entorno en producción. En general, todos los resultados son mejores que los resultados con el modelo base.

Si se comparan las arquitecturas entre sí, se puede ver el resultado que se esperaba. Los modelos densos son los peores de los cuatro, a pesar de que sus resultados son bastante buenos de por sí. Usando una arquitectura que pueda aprender también del pasado se pueden mejorar ligeramente los resultados respecto al modelo denso. Estas mejorías son notables en los modelos que usan ventanas pequeñas. Por el contrario, a mayor tamaño de ventanas las SRNN comienzan a rendir ligeramente peor que las densas. El motivo de dicho patrón puede darse debido a que estos modelos tienen demasiada información y no pueden sintetizar información muy lejana en el pasado. Este es el principal problema de las SRNN que se discutía en la Sección 2.6.1.

En cuanto a las arquitecturas LSTM cabe mencionar que los resultados calculados generan una excepción en cuanto a que todos los modelos actúan mejor que el modelo base. En concreto, cuando se hace uso de arquitecturas LSTM para una predicción de único intervalo, es decir, los tamaños de ventana son: (3, 1), (5, 1), (8, 1), (12, 1) o (24, 1), los resultados obtenidos son pésimos.

Finalmente, el modelo Autoregresivo es el modelo que mejor resultados genera y el más regular al cambio de configuración de tamaños de ventanas y existe poca diferencia entre los modelos que menos predicen (3, 1) frente a los modelos que abarcan más intervalos para predecir ((48, 12) o (48, 24)) siendo este el modelo que mejores resultados obtiene en general.

Otro aspecto a mencionar y que es común para la mayoría de los modelos es que a medida que aumentan la cantidad de intervalos a predecir, el error también aumenta ligeramente en un grado que depende del modelo que se esté evaluando. Este aumento es tan pequeño que se podría concluir que los resultados obtenidos para todas las ventanas son los mismos para todas las configuraciones. Por lo cual, daría igual usar una ventana de tamaño (8, 3) que una ventana (36, 12); Las métricas son prácticamente iguales.

7.2. Conocimientos adquiridos

Durante el desarrollo de este trabajo he podido adquirir distintos conocimientos. Antes, de comenzar con el proyecto mi principal objetivo personal con este proyecto era poder entender las matemáticas que había detrás de una red neuronal y como funcionaba el algoritmo de *Backpropagation* a nivel práctico. Sin duda alguna, no solo he sido capaz de aprender dichos conceptos sino que además he aprendido a como usar distintas herramientas y librerías para poder trabajar con redes neuronales.

Bien es cierto que no hace falta entender el funcionamiento de una red neuronal para poder usar una, pero también es cierto que con limitados conocimientos provoca que no se obtengan los mejores resultados. Por esa razón, decidí embarcarme en este proyecto y sinceramente creo que he

superado mis expectativas y he conocido nuevas fascinantes ramas sobre la informática que espero que en el futuro pueda seguir estudiándolas y usándolas en la práctica.

Obviamente, el querer aprender todo lo básico sobre redes neuronales no quiere decir que quiera desarrollar toda la parte práctica de nuevo cuando librerías como *Keras* o *Tensorflow* ya ofrecen un uso simplificado de estas. Es por eso, que durante el desarrollo de este proyecto he tenido que investigar parte de estas librerías y me he llevado una grata sorpresa al encontrarme la cantidad de facilidades que ofrece a pesar de solo conocer una parte pequeña de estas. Junto con mi desarrollo y mi aprendizaje dentro de la IA, seguiré usando y mejorando mis habilidades con estas herramientas o incluso investigar otras parecidas como puedan ser *PyTorch*.

Por último, quiero mencionar que el desarrollo del presente documento ha sido desarrollado en su totalidad con *LATEXy* ha sido una gran decisión por las facilidades que aporta puesto que el usuario solo se debe de preocupar de plasmar información. Tareas triviales como el control de índices, numeración de páginas o gestión de bibliografía es completamente ajeno al usuario.

A nivel personal, este proyecto me ha ayudado a sentar las bases de lo que es la actividad de investigación y desarrollo, siendo tú el propio dueño de tu tiempo y del trabajo que se quiere realizar y no siguiendo un enunciado y mostrando los resultados de un modo específico. He conseguido resolver dudas que se me planteaban antes de realizar el proyecto y ahora entiendo más del trabajo y las innovaciones que cada día se publican. Al mismo tiempo muchas más dudas me han surgido y espero seguir resolviendo estas dudas en el futuro continuando esta labor de investigación.

7.3. Conclusiones en cuanto a la IA

Sin duda alguna, los resultados obtenidos por lo general son bastante mejores que el modelo base y demuestra que el trabajo realizado no ha sido en vano. Es comprensible que las redes neuronales se hayan convertido en la familia de algoritmos del *Machine Learning* más populares de esta última década. Tienen un gran potencial para poder resolver problemas de distinta índole obteniendo buenos resultados. Pero las redes neuronales a su vez tienen un coste y es que el ser humano no sea capaz de interpretar como se ajustan los pesos de las redes neuronales para poder estudiar su comportamiento, el por qué de las decisiones y valores calculados y optimizar aún más los modelos. Esta deficiencia puede ser importante dependiendo de qué problema se esté resolviendo. En el proyecto que se ha presentado en la presente tesis, estos ajustes son indiferentes, puesto que no se quiere interpretar como se ha ajustado el modelo por ningún motivo: Simplemente funciona.

Pero hay otros problemas que quizás necesiten saber el por qué de los resultados obtenidos. Un ejemplo claro es el accidente ocurrido en 2018 [54] debido tanto a un despiste humano como un error de la red neuronal que conducía el coche. Si bien, la responsabilidad final fue del conductor que en el momento del accidente estaba prestando atención al móvil en vez a la carretera y era el máximo responsable y cuya principal tarea era la de supervisar las decisiones del vehículo autónomo, demuestra que este tipo de accidentes seguirán ocurriendo en el futuro, un futuro en el cual quizás no existan volantes y el usuario no pueda interactuar con el control del vehículo de forma directa [55]. Es en este punto donde surgen ciertas dudas legales ante como actuar y qué decisión judicial tomar en caso de accidente.

Está pregunta está muy relacionado a otro concepto que quizás mucha parte de la sociedad supone y es que la IA tiene como consecuencia destruir muchos puestos de trabajo en su totalidad. Y esto no es cierto completamente. Quizás muchos procesos serán sustituidos por este tipo de algoritmos pero siempre habrá alguien encargado de supervisar estas decisiones y siempre habrá un responsable de los resultados que haya obtenido una IA. Por lo que al mismo tiempo que se destruyen trabajos, otros emergen pero en menor cantidad.

Según algunos expertos la cantidad de puestos que se verán destruidos por el auge de la Inteligencia Artificial es del 30 % [56] y la creación de nuevos puestos solo será del 8 %, esta revolución tecnológica no tiene precedentes y urge nuevas medidas tanto políticas como legales para su adopción. Además, ¿está aún la sociedad lista y dispuesta a integrar este tipo de tecnologías? La IA ofrece gran cantidad de herramientas que ayudarán a la humanidad en el desarrollo de sus actividades y en el descubrimiento de nuevos inventos, conceptos y avances científicos y el gran desafío por tanto, no es solo su implantación sino también en la educación de las personas para que sepan convivir de forma cotidiana con ellas y que sean entiendan que la IA no es un enemigo, sino un aliado.

Pero el potencial de la IA y la inteligencia artificial no solo reside en sus tecnologías sino también en sus usuarios. Si confiamos (en lo esencial) en la forma en que se gestionan actualmente las sociedades, no se tendrá ninguna razón para no confiar en nosotros mismos para hacer el bien con estas tecnologías. Y si podemos suspender el presentismo y aceptar que la historia nos advierten de que no se debe jugar a ser Dios con las tecnologías poderosas, sino que estas son meramente instructivas, entonces es probable que nos liberemos de la ansiedad innecesaria sobre su uso.

La Inteligencia Artificial y el aprendizaje automático son producto tanto de la ciencia como del mito. La idea de que las máquinas podrían pensar y realizar tareas igual que los humanos tiene miles de años. Llevar los sistemas cognitivos a las máquinas tampoco son nuevas y junto con la capacidad que tendrán los modelos en superar a los humanos en ámbitos específicos hacen que estemos ante una nueva revolución tecnológica.

La mayoría de los escenarios sobre la IA del futuro son hipotéticos, pero la IA nos plantea cuestiones existenciales. Muestra que donde la ciencia se detiene, comienzan la filosofía y la espiritualidad dando de nuevo la importancia que una vez tuvieron los campos de las humanidades llevando a cabo pensamiento y conclusiones que difícilmente serán reproducibles por algún sistema artificial.

8. TRABAJO FUTURO

Gran parte de este proyecto ha sido en el estudio teórico del concepto de las redes neuronales mientras que la aplicación práctica de lo aprendido ha sido la otra parte del tiempo dedicado. En esta parte práctica, principalmente, se ha invertido tiempo tanto en el aprendizaje de las librerías que se usan en la industria como pueden ser *Tensorflow* y *Pandas*, al igual que la dedicación en el entendimiento del problema y estudio de diferentes modelos y estudiar su comportamiento. Con este trabajo por lo tanto ha quedado demostrado que el mejor modelo que ha funcionado ha sido el auto-regresivo.

Como parte del trabajo futuro se propone mejorar este modelo para minimizar todo lo posible las métricas y proponer un modelo que obtenga resultados parecidos o mejores al estado del arte actual. Para ello, se podría estudiar más en profundidad tanto los parámetros de la red como la tasa de aprendizaje, el optimizador o el error. Pero, añadiendo más información al *dataset* podría ser más útil, puesto que a mayor cantidad de datos, más preciso será el modelo. Se podrían usar por ejemplo variables como si es día festivo o la condiciones meteorológicas. Y esta es una de las principales diferencias que existen entre el *dataset* usado en este proyecto y en otros. El *dataset* usado solo contaba con tres variables sobre la información temporal: *hour*, *day_of_month* y *month* y por lo tanto, no cabe duda que añadiendo estas cantidades de variables que aporten información sobre los patrones de alquiler de bicicletas, el modelo mejorase considerablemente.

Otro aspecto a mejorar de este proyecto sería la explotación de los resultados en algún tipo de herramienta útil a la empresa o a los usuarios. Los modelos entrenados simplemente han sido usados para computar las métricas y posteriormente han sido descartados puesto que no había planes de usarlos en producción. Por el contrario, si hubiese existido algún proyecto haciendo uso de los modelos, estos se podría haber guardado en un fichero y se podría usar en diferentes herramientas. Por ejemplo, una aplicación web que indique al usuario la predicción de bicicletas disponibles para cierta hora o un software que permite a los gestores de logística reubicar las bicicletas de forma más eficiente. Estas y otras aplicaciones podrían usar como módulo principal algún modelo de red neuronal explicado en este proyecto.

to_station_id flujo en la red

Por otro lado, se podrían usar más variables conteniendo información importante como puede ser el calendario laboral o variables relacionadas con la meteorología, las cuales influyen en los patrones de comportamiento de la red de bicicletas.

Por ejemplo, un lunes no laboral, la cantidad de bicicletas alquiladas en un parque será mayor que cualquier otro lunes de otra semana. U otro ejemplo, si un día las condiciones meteorológicas nos son favorables para el uso de bicicleta, la cantidad de bicicletas alquiladas se reducirán considerablemente. Este tipo de variables son algún ejemplo que se podría añadir al *dataset* y así poder mejorar la precisión del modelo pero por falta de tiempo y por simplicidad no se han añadido. Además, los resultados obtenidos de por sí se han considerado bastante buenos y por lo tanto, no se ha visto la necesidad de invertir tiempo en este apartado aunque sería un buen estudio como mejoraría estos modelos con dichos cambios.

Finalmente, sería interesante estudiar si es factible poder usar este proyecto pero usando otros datasets y probando así su portabilidad.

9. REFERENCIAS

- [1] M. of National Development y H. VÁTI Nonprofit Ltd., «The Territorial State and Perspectives of the European Union», 2011. dirección: https://ec.europa.eu/regional_policy/sources/policy/what/territorial-cohesion/territorial_state_and_perspective_2011.pdf.
- [2] A. Infanzon, «Descifrando Enigma ayer, analítica y big data hoy», 2015. dirección: <https://www.forbes.com/descifrando-enigma-ayer-analitica-y-big-data-hoy/>.
- [3] P. Das, K. Acharjee, P. Das y V. Prasad, «VOICE RECOGNITION SYSTEM: SPEECH-TO-TEXT», *Journal of Applied and Fundamental Sciences*, vol. 1, págs. 2395-5562, nov. de 2015.
- [4] N. Li, S. Liu, Y. Liu, S. Zhao y M. Liu, «Neural Speech Synthesis with Transformer Network», *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, págs. 6706-6713, jul. de 2019. DOI: 10.1609/aaai.v33i01.33016706.
- [5] A. Gautam y S. Mohan, «A Review of Research in Multi-Robot Systems», ago. de 2012. DOI: 10.1109/ICIIInfS.2012.6304778.
- [6] W. Zhang, T. Mei, H. Liang, B. Li, J. Huang, Z. Xu, Y. Ding y W. Liu, «Research and Development of Automatic Driving System for Intelligent Vehicles», *Advances in Intelligent Systems and Computing*, vol. 215, págs. 675-684, sep. de 2014. DOI: 10.1007/978-3-642-37835-5_58.
- [7] Y. Jing, Y. Yang, Z. Feng, J. Ye, Y. Yu y M. Song, *Neural Style Transfer: A Review*, mayo de 2017.
- [8] J. Haugeland, «Artificial intelligence: The very idea (MIT Press, Cambridge, MA, 1985); 287 pp.», *Artificial Intelligence*, vol. 29, págs. 349-353, sep. de 1986.
- [9] R. Bellman, *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd Fraser Publishing Company, 1978.
- [10] E. Charniak y D. McDermott, *Introduction to Artificial Intelligence*. Addison Wesley, 1985.
- [11] P. Winston, *Artificial Intelligence*. Addison-Wesley Publishing Company, 1976.
- [12] R. Kurzweil, *The Age of Intelligent Machines*. MIT Press, 1990.
- [13] E. Rich y K. Knight, *Artificial Intelligence*. 1991.
- [14] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998, ISBN: 978-1-55860-467-4.
- [15] S. Russell y P. Norvig, *Artificial Intelligence: A Modern Approach*, 3.^a ed. Prentice Hall, 2010.

- [16] A. M. TURING, «I.—COMPUTING MACHINERY AND INTELLIGENCE», *Mind*, vol. LIX, n.º 236, págs. 433-460, oct. de 1950, ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433. eprint: <https://academic.oup.com/mind/article-pdf/LIX/236/433/30123314/lix-236-433.pdf>. dirección: <https://doi.org/10.1093/mind/LIX.236.433>.
- [17] S. Harnad, «The Turing Test is Not a Trick: Turing Indistinguishability is a Scientific Criterion», *SIGART Bull.*, vol. 3, n.º 4, págs. 9-10, oct. de 1992, ISSN: 0163-5719. DOI: 10.1145/141420.141422. dirección: <https://doi.org/10.1145/141420.141422>.
- [18] D. Poole, A. Mackworth y R. Goebel, *Computational Intelligence: A Logical Approach*. ene. de 1998, ISBN: 978-0-19-510270-3.
- [19] I. Salian, «SuperVize Me: What's the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?», 2018. dirección: <https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/#:~:text=In%20a%5C%20supervised%5C%20learning%5C%20model, and%5C%20patterns%5C%20on%5C%20its%5C%20own..>
- [20] P. Flach, *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge, 2012.
- [21] R. Logan y M. Tandoc, «Thinking in Patterns and the Pattern of Human Thought as Contrasted with AI Data Processing», *Information*, vol. 9, abr. de 2018. DOI: 10.3390/info9040083.
- [22] T. Kühne, «What is a Model?», ene. de 2004.
- [23] H. Kinsley y D. Kukiela, «Neural Network from Scratch in Python», 2020.
- [24] S. Ramón y Cajal, «Sobre las fibras nerviosas de la capa molecular del cerebro», *Histología Normal y Patológica*, ago. de 1888.
- [25] H. Dale y O. Loewi, «The Chemical Transmission of Nerve Action», 1936. DOI: <https://www.nobelprize.org/prizes/medicine/1936/loewi/lecture/>.
- [26] D. R. y De Robertis (h), *Biología celular y molecular*. El Ateneo, 1975.
- [27] C. R. Noback y R. J. Demarest, *The human nervous system: Basic principles of neurobiology*, 10.ª ed. Mc Graw Hill, 1981.
- [28] J. Fuster, «Cortex and Memory: Emergence of a New Paradigm», *Journal of Cognitive Neuroscience*, vol. 21, jul. de 2009. DOI: 10.1162/jocn.2009.21280.
- [29] S. C. Kleene, «Representation of events in nerve nets and finite automata», dic. de 1951.
- [30] M. A. Nielsen, «Neural Networks and Deep Learning», 2015.

- [31] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu y Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. dirección: <http://tensorflow.org/>.
- [32] C. Pascual, «Tutorial: Understanding Regression Error Metrics in Python», 2018. dirección: <https://www.dataquest.io/blog/understanding-regression-error-metrics/>.
- [33] G. Viswanathan, «Huber Error», 2020. dirección: <https://medium.com/@gobiviswaml/huber-error-loss-functions-3f2ac015cd45>.
- [34] M. Minsky y S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.
- [35] D. E. Rumelhart, G. E. Hinton y R. J. Williams, «Learning Representations by Back-propagating Errors», *Nature*, vol. 323, n.º 6088, págs. 533-536, 1986. DOI: [10.1038/323533a0](https://doi.org/10.1038/323533a0). dirección: <http://www.nature.com/articles/323533a0>.
- [36] J. Kiefer y J. Wolfowitz, «Stochastic Estimation of the Maximum of a Regression Function», *Annals of Mathematical Statistics*, vol. 23, págs. 462-466, 1952.
- [37] D. P. Kingma y J. Ba, «Adam: A Method for Stochastic Optimization», *CoRR*, vol. abs/1412.6980, 2015.
- [38] M. D. Zeiler, «ADADELTA: An Adaptive Learning Rate Method», *ArXiv*, vol. abs/1212.5701, 2012.
- [39] J. C. Duchi, E. Hazan e Y. Singer, «Adaptive Subgradient Methods for Online Learning and Stochastic Optimization», *J. Mach. Learn. Res.*, vol. 12, págs. 2121-2159, 2010.
- [40] N. Boufidis, A. Nikiforidis, K. Chrysostomou y G. Aifadopoulou, «Development of a station-level demand prediction and visualization tool to support bike-sharing systems' operators», *Transportation Research Procedia*, vol. 47, págs. 51-58, 2020, 22nd EURO Working Group on Transportation Meeting, EWGT 2019, 18th – 20th September 2019, Barcelona, Spain, ISSN: 2352-1465. DOI: <https://doi.org/10.1016/j.trpro.2020.03.072>. dirección: <http://www.sciencedirect.com/science/article/pii/S2352146520302593>.
- [41] Kaggle, *Shared bikes demand forecasting Competition*, 2020. dirección: <https://www.kaggle.com/c/shared-bikes-demand-forecasting/>.
- [42] Divvy trip history data, 2020. dirección: <https://divvy-tripdata.s3.amazonaws.com/index.html>.

- [43] *Divvy station map | Chicago city | Data portal*, 2020. dirección: <https://data.cityofchicago.org/Transportation/Divvy-station-map/89n3-p56s>.
- [44] *Weather history download Chicago*, 2020. dirección: https://www.meteoblue.com/en/weather/archive/export/chicago_united-states-of-america_4887398.
- [45] G. Van Rossum y F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009, ISBN: 1441412697.
- [46] F. Chollet y col. (2015). «Keras», dirección: <https://github.com/fchollet/keras>.
- [47] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke y T. E. Oliphant, «Array programming with NumPy», *Nature*, vol. 585, n.º 7825, págs. 357-362, sep. de 2020. DOI: 10.1038/s41586-020-2649-2. dirección: <https://doi.org/10.1038/s41586-020-2649-2>.
- [48] T. pandas development team, *pandas-dev/pandas: Pandas*, ver. latest, feb. de 2020. DOI: 10.5281/zenodo.3509134. dirección: <https://doi.org/10.5281/zenodo.3509134>.
- [49] J. D. Hunter, «Matplotlib: A 2D graphics environment», *Computing in Science & Engineering*, vol. 9, n.º 3, págs. 90-95, 2007. DOI: 10.1109/MCSE.2007.55.
- [50] J. D. Garrett, «SciencePlots (v1.0.6)», feb. de 2021. DOI: 10.5281/zenodo.4106650. dirección: <http://doi.org/10.5281/zenodo.4106650>.
- [51] R. Matthew Neil; Stones, *The Linux Environment*. Indiana, US, 2008, ISBN: 978-0-470-14762-7.
- [52] Reddit, *How do you represent time-of-day in artificial neural networks?*, 2014. dirección: https://www.reddit.com/r/MachineLearning/comments/lutxnk/how_do_youRepresent_timeofday_in_artificial/.
- [53] *Time series forecasting*, 2020. dirección: https://www.tensorflow.org/tutorials/structured_data/time_series.
- [54] T. GRIGGS y D. WAKABAYASHI, «How a Self-Driving Uber Killed a Pedestrian in Arizona», *The New York Times*, mar. de 2018. dirección: <https://www.nytimes.com/interactive/2018/03/20/us/self-driving-uber-pedestrian-killed.html>.
- [55] A. J. Hawkins, «GM will make an autonomous car without steering wheel or pedals by 2019», *The Verge*, ene. de 2018. dirección: <https://www.theverge.com/2018/1/12/16880978/gm-autonomous-car-2019-detroit-auto-show-2018>.

- [56] «Jobs lost, jobs gained: What the future of work will mean for jobs, skills, and wages», *McKinsey Global Institute*, jul. de 2020. dirección: <https://www.mckinsey.com/featured-insights/future-of-work/jobs-lost-jobs-gained-what-the-future-of-work-will-mean-for-jobs-skills-and-wages#>.

A. VARIABLES MATEMÁTICAS

Símbolo Descripción

L_i Capa i de la red neuronal

$L_{i,j}$ Neurona i de la capa i de la red neuronal

x Vector de entrada del modelo

$[1] \oplus x]^T$

b Sesgo de una neurona

$b^{L_{i,j}}$ Sesgo de la neurona j en la capa i

w Vector de pesos

w^* Nuevo vector de pesos tras el calculo con ∇f

$w^{L_{i,j}}$ Vector de pesos para la neurona j de la capa i

W^{L_i} Matriz de pesos de la capa i formada también por los valores b

W^{L_i} Matriz de pesos de la capa i formada también por los valores b

W^* Nueva matriz de pesos tras el calculo con ∇f

z Vector escalar entre w e y o w y x

z^{L_i} Vector escalar entre w^{L_i} e $y^{L_{i-1}}$ o w^{L_1} y x

$a()$ Función de activación

$a()^{L_i}$ Función de activación para la capa i

$a'()$ Derivada de la función de activación

$a'()^{L_i}$ Derivada de la función de activación para la capa i

y Vector de salida del modelo

y^{L_i} Vector de salida para la capa L_i

\hat{y} Vector real

\hat{y}_i Vector real que se encuentra en la posición i del *dataset*

$c()$ Función de coste

$c'()$ Derivada de la función de coste

∇f Vector gradiente

∇f_b Valor del vector gradiente para b

∇f_w Subvector del vector gradiente para w

∂ Derivada parcial

$\frac{\partial a}{\partial b}$ Derivada parcial de a respecto a b

η Tasa de aprendizaje

\oplus Concatenación de vectores

B. MUESTRA DEL DATASET ORIGINAL DE DIVVY

start_time	end_time	tripduration	from_station_id	from_station_name	to_station_id	to_station_name	gender	birthyear
2019-09-23 16:48:09	2019-09-23 17:01:57	828.0	44	State St & Randolph St	21	Aberdeen St & Jackson Blvd	NaN	NaN
2019-08-19 18:18:41	2019-08-19 18:29:20	639.0	287	Franklin St & Monroe St	110	Dearborn St & Erie St	Male	1990.0
2019-01-08 08:42:22	2019-01-08 08:53:58	696.0	66	Clinton St & Lake St	161	Rush St & Superior St	Male	1971.0
2019-05-18 13:27:52	2019-05-18 13:53:41	1,549.0	94	Clark St & Armitage Ave	35	Street Dr & Grand Ave	Male	1988.0
2019-03-01 07:04:37	2019-03-01 07:12:09	452.0	174	Canal St & Madison St	48	Larrabee St & Kingsbury St	Male	1966.0
2019-03-13 18:13:48	2019-03-13 18:17:25	217.0	210	Ashland Ave & Division St	130	Damen Ave & Division St	Male	1995.0
2019-08-11 20:51:13	2019-08-11 20:14:25	552.0	93	Sheffield Ave & Willow St	143	Sedgwick Ave & Webster Ave	Female	1995.0
2019-01-24 16:50:10	2019-01-24 17:21:20	1,870.0	525	Glenwood Ave & Touhy Ave	243	Lincoln Ave & Sunnyside Ave	Female	1986.0
2019-07-27 13:19:12	2019-07-27 13:33:34	862.0	72	Wabash Ave & 16th St	197	Michigan Ave & Madison St	Male	1995.0
2019-07-31 22:52:44	2019-07-31 22:22:41	1,796.0	52	Michigan Ave & Cortland St	58	Marsalis Ave & Kingsbury St	NaN	NaN
2019-05-14 10:38:31	2019-05-14 10:44:45	374.0	236	Sedgwick St & Schiller St	48	Larrabee St & Kingsbury St	Male	1986.0
2019-06-16 19:25:32	2019-06-16 19:40:59	927.0	85	Michigan Ave & Oak St	224	Haled St & Willow St	Male	1988.0
2019-06-10 18:18:33	2019-06-10 18:24:33	360.0	38	Clark St & Lake St	197	Michigan Ave & Madison St	Male	1993.0
2019-12-12 09:55:38	2019-12-12 10:21:01	1,522.0	296	Broadway & Belmont Ave	69	Damen Ave & Pierce Ave	NaN	NaN
2019-09-09 09:09:44	2019-09-09 09:25:01	1,036.0	26	McClurg Ct & Illinois St	620	Orleans St & Chestnut St (NEAT Ap)	Female	1992.0
2019-07-07 09:15:21	2019-07-07 10:26:40	4,279.0	99	Lake Shore Dr & Ohio St	6	Dusable Harbor	Female	1992.0
2019-11-26 08:21:39	2019-11-26 08:34:49	790.0	376	Artesian Ave & Hubbard St	217	Elizabeth (May) St & Fulton St	Female	1980.0
2019-07-22 08:09:05	2019-07-22 08:16:59	473.0	138	Clybourn Ave & Division St	133	Kingsbury St & Kinzie St	Male	1984.0
2019-05-10 18:43:10	2019-05-10 18:51:40	510.0	99	Lake Shore Dr & Ohio St	180	Ritchie Cl & Banks St	NaN	NaN
2019-04-05 09:38:21	2019-04-05 09:48:30	609.0	46	Wells St & Walton St	287	Franklin St & Monroe St	Male	1992.0
2019-07-22 16:50:47	2019-07-22 17:01:03	616.0	91	Clinton St & Washington Blvd	19	Throop (Loomis) St & Taylor St	Male	1997.0
2019-06-03 21:09:36	2019-06-03 21:14:00	1,884.0	276	California Ave & North Ave	124	Damen Ave & Sunnyside Ave	Female	1976.0
2019-08-01 19:20:36	2019-08-01 19:40:00	1,164.0	130	Damen Ave & Division St	124	Elizabeth (May) St & Fulton St	Female	1980.0
2019-07-17 16:24:52	2019-07-17 16:39:45	893.0	157	Lake Shore Dr & Wellington Ave	268	Kingsbury St & Kinzie St	Male	1997.0
2019-10-09 15:58:44	2019-10-09 16:09:00	616.0	129	Blue Island Ave & 18th St	281	Lake Shore Dr & North Blvd	Male	1988.0
2019-04-08 11:34:37	2019-04-08 11:46:25	708.0	25	Michigan Ave & Pearson St	196	Western Ave & 24th St	Male	1965.0
2019-06-11 17:14:37	2019-06-11 18:13:44	1,966.0	43	Michigan Ave & Washington St	254	Cityfront Plaza Dr & Pioneer Ct	Male	1965.0
2019-05-25 05:54:39	2019-05-25 06:04:43	1,014.0	66	Clinton St & Lake St	97	Pine Grove Ave & Irving Park Rd	NaN	NaN
2019-08-18 11:29:24	2019-08-18 11:51:34	1,330.0	199	Wabash Ave & Grand Ave	161	Field Museum	Male	1976.0
2019-09-10 08:31:13	2019-09-10 08:37:56	403.0	85	Michigan Ave & Oak St	194	Rush St & Superior St	Female	1964.0
2019-10-25 15:59:12	2019-10-25 16:07:03	471.0	211	St. Clair St & Erie St	194	Webash Ave & Wacker Pl	Male	1993.0
2019-08-13 16:01:17	2019-08-13 16:05:56	279.0	121	Blackstone Ave & Hyde Park Blvd	418	Orientes St & Hubbard St	Female	1981.0
2019-07-08 20:05:44	2019-07-08 20:13:38	473.0	156	Clark St & Wellington Ave	131	Ellis Ave & 53rd St	Male	1966.0
2019-08-11 13:52:22	2019-08-11 13:52:39	1,014.0	94	Clark St & Armitage Ave	289	Lincoln Ave & Belmont Ave	Male	1984.0
2019-03-12 13:50:21	2019-03-12 13:54:03	222.0	32	Racine Ave & Congress Pkwy	22	Wells St & Concord Ln	Male	1990.0
2019-12-07 08:29:06	2019-12-07 08:46:15	1,028.0	141	Clark St & Lincoln Ave	347	Rush St & Superior St	Female	1993.0
2019-10-03 18:58:44	2019-10-03 19:22:45	1,440.0	365	Haled St & North Branch St	117	Ashtland Ave & Grace St	Male	1991.0
2019-12-29 10:48:05	2019-12-29 10:58:48	642.0	299	Michigan Ave & Roscoe St	220	Orleans St & Hubbard St	Male	1991.0
2019-07-07 00:04:09	2019-07-07 00:24:27	1,218.0	291	Wells St & Evergreen Ave	7	Ellis Ave & Belmont Ave	Male	1991.0
2019-11-11 06:27:59	2019-11-11 06:33:24	1,764.0	94	Clark St & Armitage Ave	67	Field Blvd & South Water St	Male	1991.0
2019-04-13 19:09:09	2019-04-13 19:16:26	1,941.0	190	Racine Ave & Huron St(*)	23	Sheffield Ave & Fullerton Ave	Female	1976.0
2019-09-11 07:52:42	2019-09-11 08:05:43	781.0	331	Haled St & Clarendon Ave	48	Orleans St & Elm St (*)	Female	1979.0
2019-07-10 18:28:05	2019-07-10 18:57:29	1,764.0	29	Lake Shore Dr & North Blvd	29	Lake Shore Dr & Kingsbury St	Male	1986.0
2019-08-09 10:48:54	2019-08-09 12:06:15	4,640.0	623	Michigan Ave & 8th St	268	Noble St & Milwaukee Ave	Male	1996.0
2019-08-08 05:39:47	2019-08-08 05:45:06	319.0	283	LaSalle St & Jackson Blvd	47	Clark St & Drummond Pl	Male	1983.0
2019-04-08 17:48:11	2019-04-08 18:04:44	993.0	144	Southport Ave & Roscoe St	7	Field Blvd & South Water St	Male	1983.0
2019-07-07 07:51:29	2019-07-07 08:06:40	910.0	90	Larrabee St & Webster Ave	230	State St & Kinzie St	Male	1984.0
2019-11-29 14:14:22	2019-11-24 12:03:18	359.0	68	Wilton Ave & Diversey Pkwy	359	Lincoln Ave & Roosevelt Ave	Female	1989.0
2019-11-22 09:42:12	2019-11-22 10:16:53	369.074.0	343	Racine Ave & Wrightwood Ave	67	Larrabee St & Division St	Female	1991.0
2019-07-25 16:56:04	2019-07-25 16:58:51	1,640.0	131	Lincoln Ave & Belmont Ave	283	Sheffield Ave & Fullerton Ave	Female	1995.0
2019-04-13 09:13:59	2019-04-13 09:16:34	155.0	229	Franklin St & Jackson Blvd	68	LaSalle St & Jackson Blvd	Male	1976.0
2019-04-08 17:48:11	2019-04-08 17:57:39	2,062.0	90	Southport Ave & Roscoe St	227	Clinton St & Tilden St	Male	1992.0
2019-04-05 16:46:32	2019-04-05 17:00:32	840.0	138	Millennium Park	35	Southport Ave & Waveland Ave	Female	1984.0
2019-11-24 20:31:18	2019-11-24 20:08:39	320.0	419	Clinton St & Tilden St	322	Street Dr & Grand Ave	Female	1993.0
2019-06-20 18:23:50	2019-06-20 18:42:10	465.0	131	Clybourn Ave & Division St	90	Clybourn Ave & Division St	Male	1995.0
2019-06-14 16:29:10	2019-06-14 16:40:44	694.0	182	Kimball Ave & 53rd St	283	Kimball Ave & Division Ave	Male	1981.0
2019-04-26 14:36:59	2019-04-26 15:31:05	3,246.0	577	Wells St & Elm St	329	Millennium Park	Male	1981.0
2019-06-10 17:36:08	2019-06-10 17:50:38	890.0	283	Stony Island Ave & South Chicago Ave	11	Lake Shore Dr & Diversey Pkwy	Male	1992.0
2019-08-18 14:41:50	2019-08-18 14:51:00	549.0	330	LaSalle St & Jackson Blvd	182	Jeffery Blvd & 71st St	Female	1996.0
2019-05-03 06:07:25	2019-05-03 06:16:04	519.0	23	Lincoln Ave & Addison St	166	Wells St & Elm St	Male	1988.0
						Orleans St & Elm St (*)	Male	1977.0

Tabla 8: Muestras aleatorias del dataset original de Divvy

¹Se omiten las columnas: *bikeid*, *tripid* y *usertype*

C. MUESTRA DEL DATASET DE INTERVALOS

start_time	hour	day_of_week	month	quantity_1	quantity_2	quantity_3	quantity_4	quantity_631	quantity_632
2019-09-22 03:00:00	3	5	9	0.0	0.0	0.0	0.0	0.0	0.0
2017-01-10 00:00:00	0	2	1	0.0	0.0	1.0	0.0	0.0	0.0
2019-12-01 01:00:00	1	5	12	1.0	0.0	0.0	0.0	0.0	0.0
2018-01-02 17:00:00	17	1	1	0.0	0.0	0.0	0.0	0.0	0.0
2018-11-01 14:00:00	14	3	11	0.0	0.0	4.0	1.0	0.0	0.0
2017-01-23 03:00:00	3	1	1	0.0	0.0	0.0	0.0	1.0	0.0
2018-11-12 16:00:00	16	7	11	0.0	0.0	0.0	0.0	0.0	1.0
2019-12-25 12:00:00	12	1	12	0.0	0.0	0.0	0.0	0.0	0.0
2019-02-01 00:00:00	0	3	2	0.0	0.0	0.0	0.0	0.0	0.0
2017-03-04 08:00:00	8	6	3	0.0	0.0	0.0	1.0	0.0	0.0
2018-03-25 20:00:00	20	6	3	1.0	0.0	0.0	0.0	1.0	2.0
2019-06-08 07:00:00	7	4	6	3.0	4.0	0.0	4.0	0.0	0.0
2017-09-30 19:00:00	19	6	9	2.0	8.0	2.0	1.0	2.0	0.0
2018-08-22 21:00:00	21	2	8	0.0	0.0	2.0	1.0	3.0	0.0
2017-02-22 08:00:00	8	3	2	0.0	0.0	0.0	0.0	0.0	1.0
2017-07-05 18:00:00	18	3	7	1.0	3.0	17.0	10.0	1.0	3.0
2019-07-21 11:00:00	11	5	7	0.0	1.0	5.0	5.0	0.0	0.0
2019-07-27 00:00:00	0	4	7	1.0	0.0	0.0	0.0	0.0	0.0
2018-12-09 21:00:00	21	6	12	0.0	0.0	0.0	0.0	0.0	0.0
2018-04-16 06:00:00	6	7	4	1.0	0.0	0.0	0.0	0.0	0.0
2019-02-27 13:00:00	13	1	2	0.0	1.0	3.0	0.0	0.0	0.0
2019-10-25 02:00:00	2	3	10	0.0	0.0	0.0	0.0	0.0	0.0
2017-12-26 18:00:00	18	2	12	0.0	0.0	0.0	0.0	0.0	0.0
2017-10-29 07:00:00	7	7	10	2.0	0.0	0.0	0.0	0.0	1.0
2018-01-20 11:00:00	11	5	1	0.0	0.0	1.0	1.0	2.0	1.0
2018-06-08 03:00:00	3	4	6	0.0	0.0	0.0	0.0	0.0	0.0
2019-10-29 06:00:00	6	7	10	0.0	0.0	0.0	0.0	0.0	0.0
2019-11-25 11:00:00	11	6	11	3.0	0.0	2.0	8.0	0.0	0.0
2017-06-06 09:00:00	9	2	6	1.0	3.0	1.0	10.0	2.0	2.0
2018-03-16 09:00:00	22	4	3	0.0	3.0	0.0	0.0	0.0	0.0
2019-11-06 10:00:00	10	1	11	0.0	4.0	0.0	0.0	0.0	0.0
2017-06-11 17:00:00	17	7	6	1.0	11.0	21.0	10.0	4.0	3.0
2018-07-21 18:00:00	18	5	7	0.0	0.0	3.0	2.0	0.0	0.0
2019-05-25 11:00:00	11	4	5	0.0	0.0	6.0	0.0	1.0	1.0
2018-03-02 01:00:00	1	4	3	5.0	0.0	0.0	0.0	0.0	0.0
2018-12-21 06:00:00	6	4	12	0.0	0.0	0.0	0.0	0.0	0.0
2019-01-07 06:00:00	6	6	1	0.0	0.0	0.0	0.0	0.0	0.0
2018-05-04 13:00:00	13	4	5	1.0	0.0	5.0	2.0	3.0	3.0
2019-01-25 09:00:00	9	3	1	0.0	0.0	0.0	0.0	0.0	0.0
2019-11-24 19:00:00	19	5	11	0.0	0.0	0.0	0.0	0.0	0.0
2018-10-18 16:00:00	16	3	10	1.0	2.0	14.0	5.0	0.0	0.0
2018-07-14 23:00:00	23	5	7	1.0	1.0	6.0	0.0	0.0	0.0
2017-08-10 02:00:00	2	4	8	0.0	0.0	1.0	0.0	1.0	1.0
2017-04-13 00:00:00	0	4	4	0.0	0.0	0.0	0.0	3.0	0.0
2017-03-23 20:00:00	20	4	3	0.0	0.0	0.0	0.0	0.0	0.0
2018-09-04 16:00:00	16	9	3	21.0	20.0	10.0	2.0	2.0	2.0
2019-04-11 09:00:00	9	2	4	0.0	0.0	1.0	0.0	0.0	0.0
2019-05-25 01:00:00	1	4	5	0.0	0.0	0.0	0.0	0.0	0.0
2018-01-03 11:00:00	11	2	1	0.0	1.0	4.0	0.0	4.0	0.0
2018-04-19 00:00:00	0	3	4	0.0	0.0	0.0	0.0	0.0	0.0
2019-09-27 16:00:00	16	3	9	6.0	14.0	7.0	0.0	0.0	2.0
2017-02-16 01:00:00	1	4	2	0.0	0.0	0.0	0.0	0.0	0.0
2017-05-09 19:00:00	19	2	5	2.0	1.0	3.0	0.0	1.0	1.0
2019-11-26 01:00:00	1	7	11	1.0	0.0	0.0	0.0	1.0	0.0
2018-08-12 05:00:00	5	6	8	0.0	0.0	0.0	0.0	0.0	0.0

Tabla 9: Muestras aleatorias del dataset dividido por intervalos

²Se omiten las todas las estaciones cuyo *id* sea entre 5 y 631

D. CÓDIGO DE LA CLASE *WINDOWGENERATOR*

```
import tensorflow as tf
import numpy as np

class WindowGenerator():
    def __init__(self, input_width, label_width, shift,
                 train_df, val_df, test_df=None,
                 label_columns_index=-3):

        # Store the datasets
        self.train_df = train_df
        self.val_df = val_df
        self.test_df = test_df

        # Gets the indexes of the label column.
        self.label_columns_index = label_columns_index
        self.label_columns = train_df.columns[:label_columns_index]
                           .tolist()
        self.label_columns_indices = {name: i for i, name in
                                      enumerate(self.label_columns)}

        # Dict of name of column as key and index as value
        self.column_indices = {name: i for i, name in
                              enumerate(train_df.columns)}

        # Work out the window parameters.
        self.input_width = input_width
        self.label_width = label_width
        self.shift = shift

        # Handle the indexes and offsets as shown in the diagrams
        # above.
        self.total_window_size = input_width + shift

        self.input_slice = slice(0, input_width)
        self.input_indices = np.arange(self.total_window_size)[
            self.input_slice]

        self.label_start = self.total_window_size - self.label_width
        self.labels_slice = slice(self.label_start, None)
        self.label_indices = np.arange(self.total_window_size)[
            self.labels_slice]

    def split_window(self, features):
        inputs = features[:, self.input_slice, :]
        labels = features[:, self.labels_slice, :]

        if self.label_columns is not None:
            labels = tf.stack(
                [labels[:, :, self.column_indices[name]]
                 for name in self.label_columns],
                axis=-1)

        # Slicing doesn't preserve static shape information, so set
        # the shapes manually. This way the `tf.data.Datasets` are
        # easier to inspect.
        inputs.set_shape([None, self.input_width, None])
        labels.set_shape([None, self.label_width, None])
        return inputs, labels

    def make_dataset(self, data):
        if data is None:
            print("Data is None")
            return
```

```

    data = np.array(data, dtype=np.float32)
    ds = tf.keras.preprocessing.timeseries_dataset_from_array(
        data=data,
        targets=None,
        sequence_length=self.total_window_size,
        sequence_stride=1,
        shuffle=True,
        batch_size=32,)

    ds = ds.map(self.split_window)

    return ds

@property
def train(self):
    return self.make_dataset(self.train_df)

@property
def val(self):
    return self.make_dataset(self.val_df)

@property
def test(self):
    return self.make_dataset(self.test_df)

```

3

³Código basado en el tutorial oficial de Tensorflow "Time series forecasting" [31]

E. CÓDIGO DEL MODELO *AUTO-REGRESIVP*

```
import tensorflow as tf
from tensorflow.keras.layers import LSTMCell, RNN, Dense

class AutoRegressive(tf.keras.Model):
    def __init__(self, lstm_units, steps, stations):
        super().__init__()
        self.steps = steps
        self.stations = stations
        self.lstm_units = lstm_units

        # First LSTM layer
        self.lstm_cell = LSTMCell(lstm_units)
        # Also wrap the LSTMCell in an RNN to simplify the `warmup` method.
        self.lstm_rnn = RNN(self.lstm_cell, return_state=True)

        # Output layer
        self.dense = Dense(stations, activation="relu")

    def warmup(self, inputs):
        # inputs.shape => (batch, time, features)
        # x.shape => (batch, lstm_units)
        x, *state = self.lstm_rnn(inputs)

        # predictions.shape => (batch, features)
        prediction = self.dense(x)
        return prediction, state

    def call(self, inputs, training=None):
        # Use a TensorArray to capture dynamically unrolled outputs.
        predictions = []

        # Initialize the lstm state
        prediction, state = self.warmup(inputs)

        # Insert the first prediction
        predictions.append(prediction)

        # Run the rest of the prediction steps
        for n in range(1, self.steps):
            # Use the last prediction as input.
            x = prediction

            # Concat with timestamp variables
            x = tf.concat([x, inputs[:, -1, -3:]], 1)

            # Execute one lstm step.
            x, state = self.lstm_cell(x, states=state, training=training)

            # Convert the lstm output to a prediction.
            prediction = self.dense(x)

            # Add the prediction to the output
            predictions.append(prediction)

        # predictions.shape => (time, batch, features)
        predictions = tf.stack(predictions)

        # predictions.shape => (batch, time, features)
        predictions = tf.transpose(predictions, [1, 0, 2])

    return predictions
```

⁴Código basado en el tutorial oficial de Tensorflow "*Time series forecasting*" [31]

F. CÓDIGO USADO PARA COMPILAR, ENTRENAR Y EVALUAR MODELOS

```
import tensorflow as tf
import pandas as pd

from tf.keras.callbacks import EarlyStopping
from livelossplot import PlotLossesKeras


"""
Compiles and train model

@param model which will be training
@param window with the Datasets used
@param lr (learning rate) of the model
@param max_epoch for training
@param should_stop if detects overfitting the stops with a patience of 10
"""

def compile_and_fit(model, window, lr, max_epochs=150, should_stop=False):
    model.compile(
        loss=tf.losses.Huber(),
        optimizer=tf.optimizers.Adam(lr=lr),
        metrics=[MeanSquaredLogarithmicError(), MeanSquaredError(), MeanAbsoluteError(),
                 RootMeanSquaredError(), RootMeanSquaredLogarithmicError])

    # Define the callbacks for the training process
    # PlotLossesKeras plots training and validation for every epoch in real time
    # early_stopping allows to stop if detects overfitting
    cbs = [PlotLossesKeras()] + ([EarlyStopping(monitor='val_loss',
                                                patience=10,
                                                mode='min')] if should_stop else [])

    model.fit(window.train, epochs=max_epochs,
              validation_data=window.val,
              callbacks=cbs,
              verbose=2)

"""
Computes metrics and returns it as DataFrame
"""

def get_metrics(model, window):
    train_metrics = model.evaluate(window.train)
    val_metrics = model.evaluate(window.val)
    test_metrics = model.evaluate(window.test)

    metrics_names = model.metrics_names
    metrics_names[0] = "huber"
    metrics = pd.DataFrame({
        "names": metrics_names,
        "train": train_metrics,
        "val": val_metrics,
        "test": test_metrics,
    })

    return metrics

windows_sizes=[(3, 1), (5, 1), (8, 1), (8, 3), (8, 5), (12, 1),
               (12, 3), (12, 5), (24, 1), (24, 3), (24, 5)]


"""
Given a model it will train with different windows sizes
```

```

@param model which will be training
@param max_epoch for training
@param lr (learning rate) of the model
@param windows_sizes a list of tuples containing window sizes as (input, output)
'''

def model_generator(model, lr, max_epochs=150, windows_sizes=windows_sizes):
    # Loads and splits the DataFrames
    df = load_dataset()
    train_df, val_df, test_df = split_dataset(df)

    # For every windo
    for w in windows_sizes:
        # Create the windoe
        input_width, steps = r
        window = WindowGenerator(input_width=input_width,
                                 label_width=steps,
                                 shift=steps,
                                 train_df=train_df,
                                 val_df=val_df,
                                 test_df=test_df)

        # Train model
        compile_and_fit(
            model, window, lr=lr, should_stop=True, max_epochs=max_epochs)

        # Save metrics
        metrics = get_metrics(model, window)
        filename = "/path/to/results/{model.get_name()}" + \
                   "-{input_width}-{steps}.csv"
        metrics.to_csv(filename)

```

SIGLAS

- AR** Autoregressivo. 3, 70, 84–86, 90–93, 98
- CO₂** Dióxido de carbono. 1
- CSV** *Comma-Separated Values*. 66, 67, 72, 75, 87
- DL** Aprendizaje profundo. 42, 66
- EOC** En Otro Caso. 28
- IA** Inteligencia Artificial. 5–9, 30, 42, 65, 66, 99, 100
- LR** Tasa de aprendizaje. 107
- LSTM** *Long-Short Term Memory*. 3, 4, 60, 61, 63, 84–86, 90–93, 98
- MAE** Error Absoluto Medio. 33–37, 50, 88, 92
- MAPE** Porcentaje medio de error absoluto. 36, 37
- ML** *Machine Learning*. 7–9, 99
- MPE** Porcentaje medio de error. 36, 37
- MSE** Error cuadrático medio. 34–36, 38, 39, 50, 89, 91
- MSLE** Raíz del error medio cuadrático. 35, 89, 91
- NLP** Procesado del Lenguaje Natural. 38, 54
- NN** Red neuronal. 8
- OLS** Mínimo Cuadrado Ordinarios. 38, 39
- PSS** *Product Service System*. 1
- ReLU** Rectificador de Unidad Lineal. 18, 20, 60
- RMSE** Root Mean Squared Error. 35, 89, 92
- RMSLE** Error logarítmico medio cuadrático. 89, 93
- RMSProp** Root Mean Square Propagation. 49
- RNN** Red neuronal recurrente. 3, 54–56, 58, 86, 90–93
- SGD** Descenso de gradiente estocástico. 49
- SQL** *Structured Query Language*. 67
- SRNN** Red neuronal recurrente simple. 98
- SVM** Support Vector Machine. 8
- XOR** Exclusive Or. 13–15