APRENDIZAJE AUTOMÁTICO PARA SERIES TEMPORALES MULTIVARIANTES

Javier Pérez Manzano

APRENDIZAJE AUTOMÁTICO PARA SERIES TEMPORALES MULTIVARIANTES

Javier Pérez Manzano

Memoria presentada como parte de los requisitos para la obtención del título de Grado en Matemáticas por la Universidad de Sevilla.

Tutorizada por

Prof. Rafael Pino Mejías

Índice general

English Abstract

This project aims to investigate and compare different neural network architectures for multivariate time series prediction. Specifically, recurrent neural networks (RNN) and convolutional neural networks (CNN) have been widely used in this area, but new architectures such as long short-term memory (LSTM) and gated recurrent unit (GRU) networks. The goal is to develop accurate prediction models using these architectures and compare their performance.

1 Introducción

1.1 Contexto y motivación

La predicción de series temporales multivariantes es una tarea importante en muchas áreas, como la meteorología, la economía, la ingeniería, la medicina y las ciencias sociales . En estos campos, la predicción precisa de series temporales puede ayudar a tomar decisiones informadas y planificar estrategias futuras.

En las últimas décadas, el aprendizaje profundo ha demostrado ser una técnica efectiva para el modelado de series temporales. En particular, las redes neuronales recurrentes (RNN) y las redes neuronales convolucionales (CNN) han sido ampliamente utilizadas para la predicción de series temporales.

La motivación detrás de este trabajo es investigar y comparar estas diferentes arquitecturas de redes neuronales para la predicción de series temporales multivariantes. Se busca evaluar su capacidad para capturar patrones, modelar relaciones complejas entre múltiples variables y generalizar a nuevas series de tiempo.

El objetivo principal del trabajo es desarrollar modelos precisos de predicción de series temporales multivariantes utilizando estas arquitecturas de redes neuronales y comparar su desempeño en diferentes conjuntos de datos.

2 Series temporales

2.1 Introducción

Una serie temporal es una secuencia de datos observados en un período de tiempo discreto y en orden cronológico. En otras palabras, es una secuencia de valores numéricos que se registran a lo largo del tiempo en intervalos específicos, como horas, días, semanas, meses o años. El objetivo del análisis de las series temporales es la abstracción de patrones o secuencias de comportamientos que permita prever la evolución futura de la variable a estudio, suponiendo que las condiciones que envuelven a los futuros datos no cambiarán con respecto a las condiciones que enmarcan las observaciones pasadas.

Podemos clasificar las series temporales en dos tipos: univariante y multivariante. Las series temporales univariantes son aquellas en las que solo se mide una variable en cada punto temporal, mientras que las series temporales multivariantes implican la medición de múltiples variables simultáneamente en cada punto temporal. Por ejemplo, una serie temporal univariante puede ser la demanda de electricidad total registrada en un año, mientras que una serie temporal multivariante podría ser la demanda eléctrica total y el PIB registrados en el mismo año. En la figura 2.1 se representa esto mismo.

Las series temporales multivariantes son importantes en muchas áreas, como la meteorología, la economía, la ingeniería, la medicina y las ciencias sociales. En estos campos, la predicción precisa de series temporales puede ayudar a tomar decisiones informadas y planificar estrategias futuras.

Por ejemplo, en meteorología, la predicción precisa de la temperatura, la humedad y otros factores climáticos puede ayudar a las autoridades a tomar medidas para pre-

venir desastres naturales, como inundaciones y sequías. En la economía, la predicción precisa de las ventas, la demanda y los precios puede ayudar a las empresas a tomar decisiones informadas sobre la producción, el inventario y la estrategia de precios. En la medicina, la predicción precisa de la evolución de las enfermedades y la salud de los pacientes puede ayudar a los médicos a planificar el tratamiento y la atención médica.

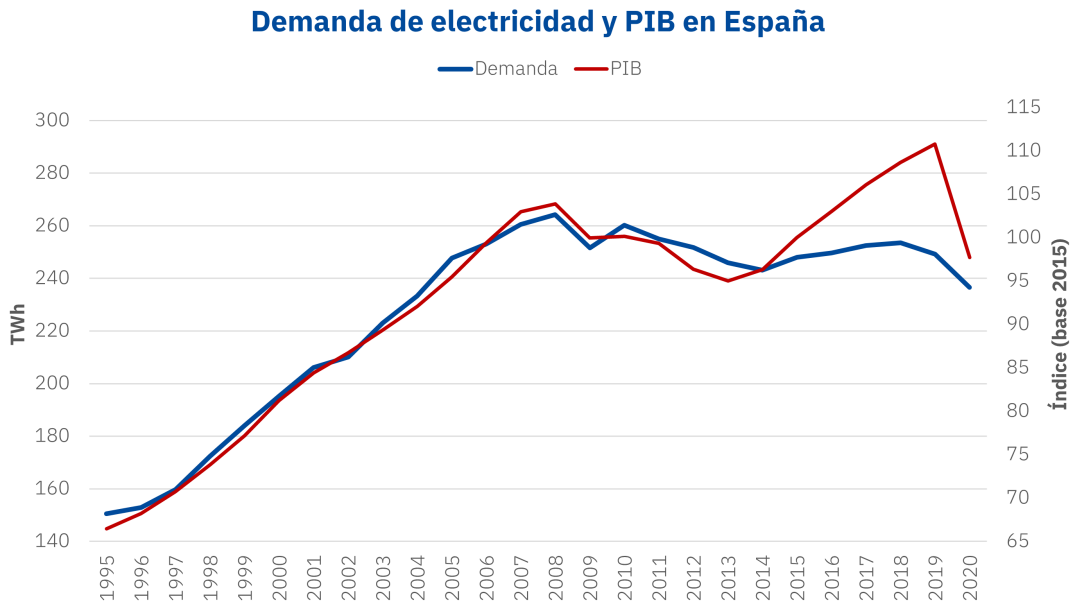


Figura 2.1: Demanda de electricidad y PIB en España

En general, la predicción precisa de series temporales multivariantes puede mejorar la toma de decisiones en una amplia gama de campos y ayudar a planificar y optimizar estrategias futuras.

2.2 Componentes de una serie temporal

Las series temporales se componen de diferentes componentes que contribuyen a la variabilidad en los datos. Estas componentes pueden descomponerse y analizarse por separado para comprender mejor el comportamiento de la serie temporal. Las componentes principales de una serie temporal son:

- Tendencia: Es la dirección general que sigue la serie temporal. Puede ser cre-

ciente, decreciente o mantenerse constante. En algunos casos, la tendencia puede ser no lineal o seguir un patrón estacional.
- Ciclos: Son fluctuaciones no periódicas que no siguen una tendencia clara. Pueden ser causados por factores económicos, políticos o sociales y pueden tener una duración variable.
- Estacionalidad: Es la variación periódica que ocurre en la serie temporal. Puede ser diaria, semanal, mensual, trimestral o anual, y puede ser causada por factores como el clima, la temporada de compras o las vacaciones.
- Ruido: Es la variación aleatoria que no se puede explicar por las otras componentes. Puede deberse a factores externos como eventos impredecibles o errores de medición.

Denotando y_t por el valor de la variable de la serie temporal en cuestión en el instante t, podemos explicar mediante la descomposición $y_t = f(T_t, E_t, R_t)$ donde f denota una función que relaciona las componentes:

- Factor de tendencia (T_t): Recoge las oscilaciones a largo plazo y las oscilaciones no periódicas (ciclos).
- Factor estacional (E_t): Contiene las fluctuaciones regulares.
- Factor residual(R_t): Caracterizado por las variaciones que no pueden ser explicadas por las componentes anteriores.

Aunque en principio podríamos definir f de múltiples formas, por su simplicidad en la práctica destacan la descomposición aditiva y la descomposición multiplicativa.

Descomposición aditiva:

$$y_t = T_t + E_t + R_t$$

Descomposición multiplicativa:

$$y_t = T_t \cdot E_t \cdot R_t$$

En ambos casos, se busca descomponer la serie temporal en sus componentes básicas para poder analizar y modelar cada una por separado y luego reconstruir la serie original. La descomposición aditiva es la mejor opción si la varianza es constante a lo largo del tiempo, mientras que en otro caso podríamos hacer uso de la descomposición aditiva.

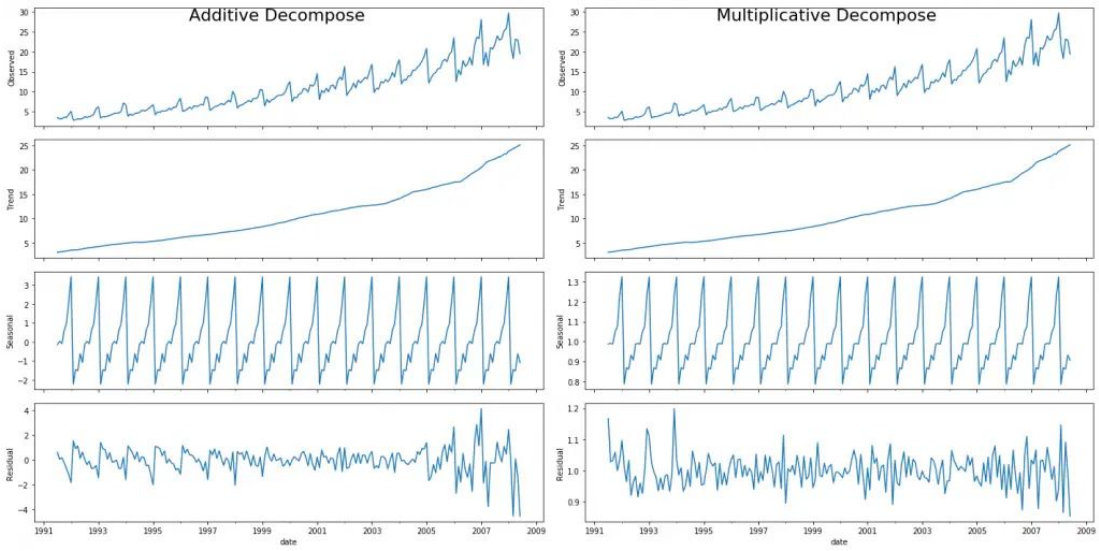


Figura 2.2: Descomposición aditiva y multiplicativa

2.3 Caracterización de una serie temporal

Una serie temporal y_t se considera estacionaria si su distribución de probabilidad conjunta no cambia en el tiempo. Es decir, la media μ_t y la varianza σ_t^2 de la serie temporal son constantes en el tiempo. Formalmente, una serie temporal se considera estacionaria si:

- La media es constante en el tiempo: $\mu_t = \mu, \forall t$
- La varianza es constante en el tiempo: $\sigma_t^2 = \sigma^2, \forall t$
- La función de autocorrelación (ACF), que mide la correlación entre una serie temporal y sus valores retrasados, depende únicamente de la distancia entre los puntos, y no de la posición en el tiempo. Esto significa que la correlación entre y_t y y_{t-k} es la misma para cualquier t y k. Formalmente, esto se expresa como:

$$\text{ACF}(k) = \text{Corr}(y_t, y_{t-k}) = \text{Corr}(y_{t+2}, y_{t+2-k}) = \cdots = \text{Corr}(y_{t+n}, y_{t+n-k})$$

En resumen, una serie temporal estacionaria es aquella en la que la media, la varianza y la correlación no cambian en el tiempo. Esto facilita la tarea de modelar y predecir la serie temporal, ya que se pueden aplicar métodos estadísticos que asumen que la serie es estacionaria.

Es importante conocer las componentes de una serie temporal para comprender su comportamiento y poder predecir futuros valores con mayor precisión. Como hemos visto, la tendencia y la estacionalidad pueden ser modeladas y eliminadas para producir una serie estacionaria, lo que hace que sea más fácil aplicar técnicas de análisis y predicción. Además, las componentes pueden ser utilizadas como variables explicativas en modelos de regresión para explicar la variación en otras variables.

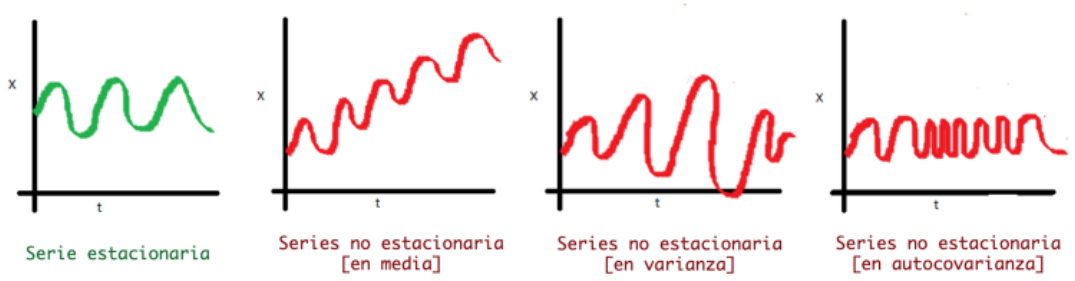


Figura 2.3: Series estacionarias y no estacionarias

Sin embargo, la idea de descomponer la variación de una serie en sus componentes básicas puede tener algunos inconvenientes. En primer lugar, la descomposición puede ser subjetiva y dependiente del usuario, lo que puede resultar en diferentes descomposiciones de la misma serie temporal. En segundo lugar, la eliminación de la tendencia y la estacionalidad puede llevar a la pérdida de información importante que puede afectar la precisión de las predicciones. Por último, en algunos casos, puede ser difícil distinguir entre la variación cíclica y la variación aleatoria, lo que puede afectar la calidad de la descomposición.

3 Redes neuronales

3.1 Introducción

El cerebro humano es uno de los órganos más complejos y fascinantes del cuerpo humano. Uno de los componentes más importantes del cerebro son las neuronas, que son células especializadas en la transmisión de señales eléctricas y químicas a través del sistema nervioso.

Las neuronas son altamente interconectadas y forman redes complejas que son responsables de procesos cognitivos como la percepción, el aprendizaje, la memoria y el pensamiento. Cada neurona recibe señales de otras neuronas y realiza cálculos en función de esas señales antes de transmitir una señal de salida.

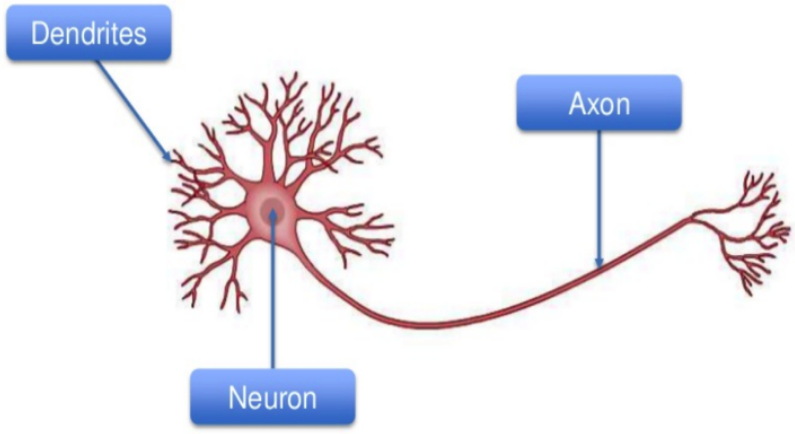


Figura 3.1: Neurona humana y sus partes principales

Tal y como se ve en la imagen 3.1, la estructura de una neurona consta de tres partes principales: el cuerpo celular o soma, las dendritas y el axón. Las dendritas son las extensiones que se ramifican desde el soma y reciben señales de entrada, mientras que el axón es una prolongación única y larga que transmite señales de salida a otras neuronas.

Las neuronas del cerebro humano son altamente adaptables y pueden cambiar sus conexiones en respuesta a la experiencia y al aprendizaje. Esto se debe en parte a la plasticidad sináptica, que es la capacidad de las conexiones neuronales para cambiar su fuerza y eficacia en función del uso.

Las redes neuronales son un modelo matemático que intenta replicar el funcionamiento del cerebro humano, mediante la interconexión de múltiples neuronas artificiales. Estas redes han experimentado un gran auge en los últimos años, gracias al crecimiento exponencial de la capacidad computacional y a la gran cantidad de datos disponibles.

El estudio de las redes neuronales se remonta a los años cuarenta, cuando Warren McCulloch y Walter Pitts propusieron el primer modelo matemático de una neurona artificial en su paper titulado "A Logical Calculus of Ideas Immanent in Nervous Activity", publicado en 1943 en el Bulletin of Mathematical Biophysics. Sin embargo, no fue hasta los años cincuenta cuando Frank Rosenblatt desarrolló el Perceptrón, el primer modelo de red neuronal capaz de aprender de forma automática. El paper principal sobre el Perceptrón es "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain" publicado en Psychological Review en 1958

En las décadas siguientes, el interés por las redes neuronales disminuyó debido a las limitaciones de la época, como la falta de datos y la capacidad computacional. Sin embargo, con el auge de la informática y la aparición de nuevas técnicas de aprendizaje automático, las redes neuronales han vuelto a estar en el centro del interés científico y tecnológico.

En la actualidad, las redes neuronales se utilizan en una gran variedad de aplicaciones, desde el reconocimiento de voz y de imágenes hasta la predicción del tiempo o la detección de fraude en el sector bancario. La capacidad de estas redes para aprender y adaptarse a nuevas situaciones las convierte en una herramienta valiosa para enfrentar los retos de la era digital.

3.2 Redes neuronales artificiales

Las redes neuronales artificiales (RNA) son modelos computacionales inspirados en las redes neuronales biológicas del cerebro humano. Estas redes están compuestas por un gran número de unidades simples llamadas neuronas artificiales, que se organizan en capas y se interconectan mediante conexiones ponderadas.

Al igual que las neuronas biológicas, las neuronas artificiales tienen la capacidad de recibir, procesar y transmitir información. La neurona artificial es el bloque básico de construcción de una red neuronal. Su funcionamiento matemático se basa en una combinación lineal de sus entradas seguida de una función de activación no lineal. Formalmente, una neurona artificial con n entradas puede ser descrita como:

$$y = f(\sum_{i=1}^{n} w_i x_i + b)$$

Donde w_i son los pesos sinápticos que se multiplican por cada entrada x_i, b es el sesgo o bias, $\sum_{i=1}^{n} w_i x_i + b$ es la suma ponderada de las entradas y f es la función de activación. El resultado de esta combinación lineal se pasa a través de la función de activación para producir la salida de la neurona, y.

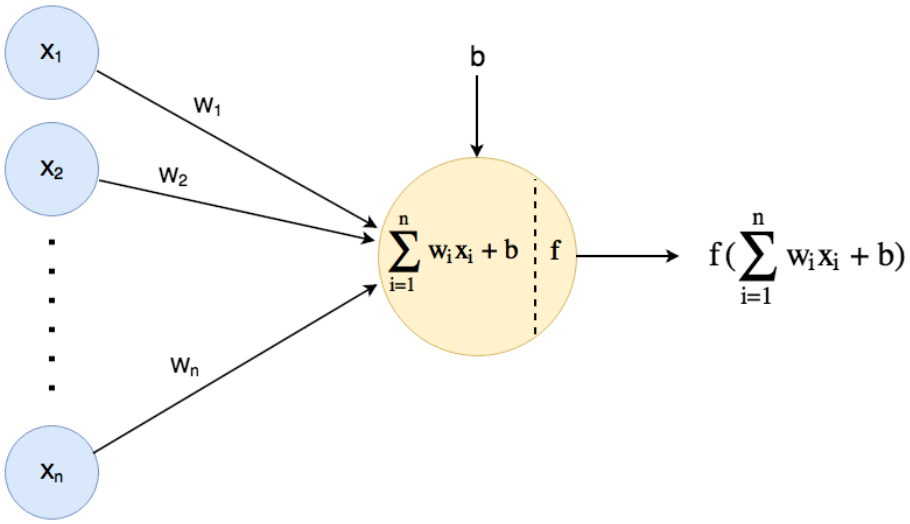
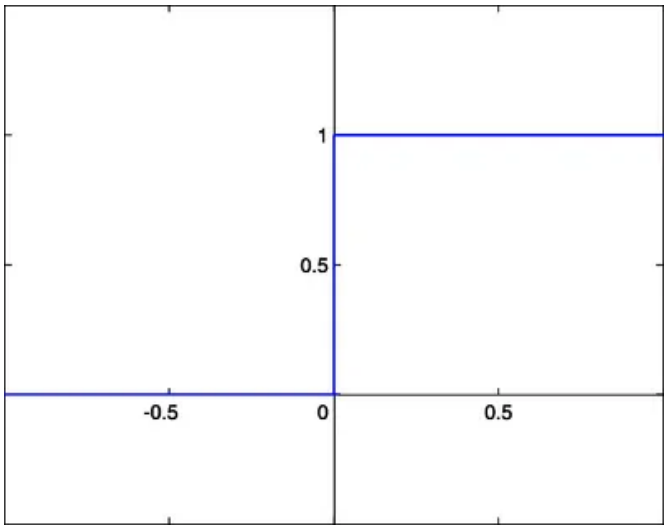


Figura 3.2: Neurona artificial

La función de activación es típicamente no lineal y se utiliza para introducir no linealidad en la respuesta de la neurona. Ejemplos de funciones de activación comunes incluyen la función escalón, la función sigmoide, la función ReLU (Rectified Linear Unit) y la función tanh (tangente hiperbólica).

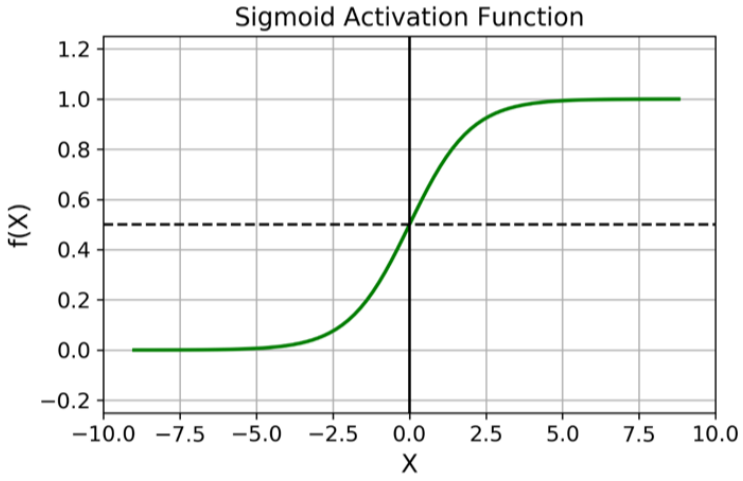
Función escalón:

$$\Theta(x) = \begin{cases} 0, & \text{si } x < 0 \\ 1, & \text{si } x \geq 0 \end{cases}$$



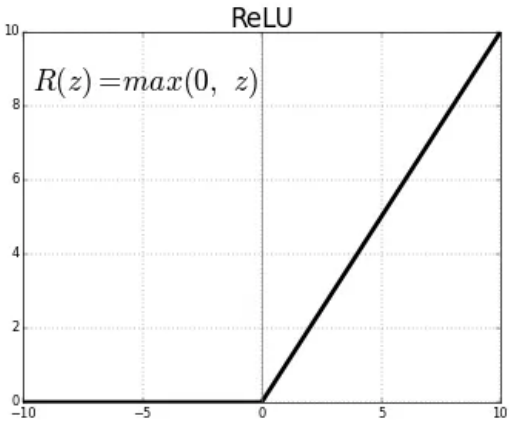
Función sigmoide:

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



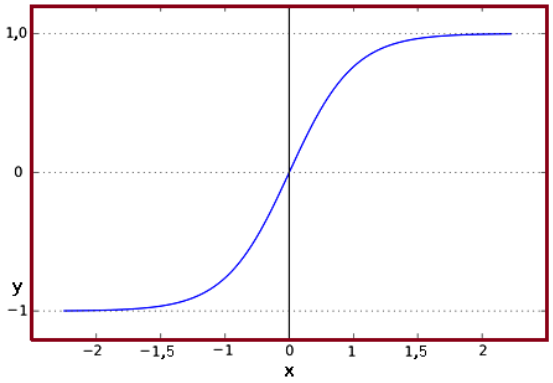
Función ReLU:

$$ReLU(x) = max(0, x)$$



Función tanh:

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



La elección de la función de activación depende del problema a resolver y de la arquitectura de la red neuronal.

En una RNA, las neuronas se agrupan en capas de entrada, ocultas y de salida. La capa de entrada recibe los datos de entrada y los transmite a la siguiente capa, la capa oculta, que realiza una serie de cálculos matemáticos en los datos de entrada y los transmite a la siguiente capa oculta o a la capa de salida.

Cada neurona en una capa oculta o de salida de una RNA utiliza una función de activación para determinar su salida, que se transmite a la siguiente capa o al resultado final.

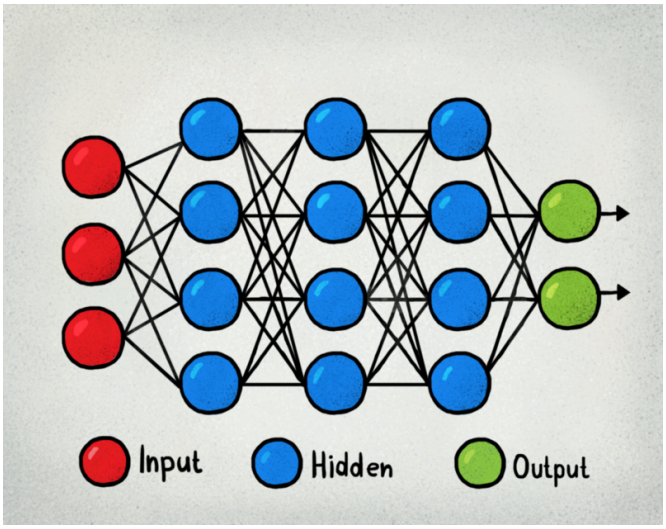


Figura 3.3: Red neuronal artificial

La salida de una capa se convierte en la entrada de la siguiente capa, y así sucesivamente, hasta que se alcanza la capa de salida, donde se produce el resultado final. En la capa de salida, la función de activación puede ser diferente de la utilizada en las capas ocultas, dependiendo de la tarea de la RNA.

A esta fase la llamaremos fase de propagación. Vamos ahora a desarrollar formalmente como se propagan los valores desde la entrada hasta la salida de una RNA con $L+1$ capas:

- Para la capa de entrada, los valores de entrada son simplemente $X = (x_1, x_2, ..., x_n) = (z_i^0, z_2^0, ..., z_n^0)$.
- Para cada capa oculta l con m_l neuronas, la salida z_i^l de cada neurona i se calcula como:

$$z_i^l = f\left(\sum_{j=1}^{m_l} w_{ij}^l z_j^{l-1} + b_i^l\right), \quad i = 1, 2, \ldots, m_l$$

  donde w_{ij}^l es el peso de la conexión entre la neurona i en la capa anterior y la neurona j en la capa oculta l, z_j^{l-1} es la salida de la neurona j en la capa anterior, b_i^l es el sesgo de la neurona i en la capa oculta l, y f es la función de activación.
- Para la capa de salida, cada $y_i = z_i^L$ se calcula como:

$$y_i = z_i^L = f\left(\sum_{j=1}^{m_L} w_{ij}^L z_j^{L-1} + b_i^L\right), \quad i = 1, 2, \ldots, m_L$$

donde y_i es la salida en la i-ésima neurona de la capa de salida, w_{ij}^L es el peso sináptico entre la i-ésima neurona de la última capa oculta y la j-ésima neurona de la capa de salida, z_j^{L-1} es la salida de la j-ésima neurona de la última capa oculta, b_i^L es el sesgo (bias) de la k-ésima neurona de la capa de salida, f es la función de activación y m_L es el número de neuronas en la última capa oculta.

Una de las principales ventajas de las RNA es su capacidad para aprender y adaptarse a partir de ejemplos de entrenamiento. Esto se logra mediante el ajuste de los pesos sinápticos durante el proceso de entrenamiento, que se realiza mediante el uso de algoritmos de aprendizaje. Existen varios algoritmos de aprendizaje, a continuación se presentan los principales:

- Aprendizaje supervisado: es un algoritmo en el que se le proporciona a la red neuronal un conjunto de datos de entrada y salida esperada. La red ajusta los pesos de las conexiones entre las neuronas para minimizar la diferencia entre la salida real y la esperada. Ejemplos de algoritmos de aprendizaje supervisado son el backpropagation y el perceptrón multicapa.
- Aprendizaje no supervisado: en este caso, la red neuronal no tiene acceso a la salida esperada. En cambio, se le proporciona solo los datos de entrada y se le pide que encuentre patrones o estructuras en ellos. Ejemplos de algoritmos de aprendizaje no supervisado son el clustering y la reducción de dimensionalidad.
- Aprendizaje por refuerzo: en este tipo de aprendizaje, la red neuronal interactúa con un entorno y recibe recompensas o penalizaciones por sus acciones. El objetivo es que la red aprenda a tomar decisiones que maximicen la recompensa a largo plazo. Ejemplos de algoritmos de aprendizaje por refuerzo son Q-Learning y la red neuronal de actor-crítico.

Cada uno de estos algoritmos tiene sus propias ventajas y desventajas, y se aplican en diferentes contextos según las necesidades del problema que se esté tratando de resolver.

3.2.1 Algoritmo de retropropagación

El algoritmo de retropropagación o backpropagation es uno de los algoritmos de aprendizaje más populares y ampliamente utilizados en redes neuronales. Su objetivo principal es ajustar los pesos sinápticos de una red neuronal para que la salida de la

red se acerque lo más posible a la salida deseada. El algoritmo de retropropagación se basa en el método de descenso de gradiente para minimizar la función de error de la red.

El algoritmo se puede dividir en dos fases: propagación hacia adelante (forward propagation) y retropropagación de errores (backpropagation). Durante la propagación hacia adelante, las entradas se introducen en la red y se propagan hacia la salida tal y como hemos visto anteriormente. Durante la retropropagación de errores, el error se calcula en la salida y se propaga hacia atrás a través de la red para ajustar los pesos sinápticos.

La retropropagación de errores se puede resumir en los siguientes pasos:

1. Inicializar los pesos sinápticos de la red neuronal de forma aleatoria o según alguna estrategia predefinida.
2. Para cada ejemplo de entrenamiento, se introduce la entrada en la red neuronal y se propaga hacia adelante para obtener la salida de la red.
3. Se calcula el error entre la salida de la red y la salida deseada. Sea $y = (y_1, y_2, ..., y_{m_L})$ la salida producida por la red y $y^* = (y_1^*, y_2^*, ..., y_{m_L}^*)$ la salida deseada. Una de las formas de calcular el error total de la red es como la suma de los errores cuadráticos medios entre la salida producida por la red y la salida deseada:

$$E = \frac{1}{2} \sum_{i=1}^{m_L} (y_i - y_i^*)^2$$

4. El error se propaga hacia atrás a través de la red neuronal, capa por capa, utilizando la regla de la cadena para calcular las contribuciones de los pesos a la salida. Sea w el vector de pesos sinápticos de la red, b los términos de sesgo y E el error total de la red, entonces la contribución de los pesos w_{ij} y b_i^l en la capa l a la salida son, respectivamente:

$$\frac{\partial E}{\partial w_{ij}^l} = \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial w_{ij}^l} = \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial a_j^l} \cdot z_j^{l-1}$$

$$\frac{\partial E}{\partial b_i^l} = \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial b_i^l} = \frac{\partial E}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial a_j^l}$$

donde $a_j^l = \sum_{j=1}^{m_l} w_{ij}^l z_j^{l-1} + b_i^l$

Las ultimas igualdades se deben respectivamente a que $\frac{\partial a_j^l}{\partial w_{ij}^l} = z_j^{l-1}$ y $\frac{\partial a_j^l}{\partial b_i^l} = 1$

5. Se actualizan los pesos sinápticos y los términos de sesgo de la red utilizando la regla de aprendizaje del descenso de gradiente. La actualización de los pesos y sesgos se realiza en la dirección opuesta al gradiente de la función de error con una tasa de aprendizaje predefinida α:

$$w_{ij}^{l} \leftarrow w_{ij}^{l} - \alpha \cdot \frac{\partial E}{\partial w_{ij}^{l}}$$

$$b_{i}^{l} \leftarrow b_{i}^{l} - \alpha \cdot \frac{\partial E}{\partial b_{i}^{l}}$$

La tasa de aprendizaje α controla el tamaño de los pasos que se dan en la dirección opuesta al gradiente.

El algoritmo se repite para un número predefinido de épocas o hasta que la función de error alcance un valor mínimo aceptable.

Aunque el algoritmo de backpropagation es el algoritmo de aprendizaje principal para las redes neuronales, la optimización de los pesos de una red neuronal es un problema de optimización no lineal que puede ser complejo y computacionalmente costoso. Por lo tanto, se han desarrollado algoritmos de optimización más sofisticados y eficientes que combinan el algoritmo de backpropagation con otros métodos de optimización.

El algoritmo que será principalmente implementado en los scripts de este proyecto es el ADAM (Adaptive Moment Estimation), que es un método de optimización de gradiente estocástico que se utiliza para actualizar los pesos sinápticos de una red neuronal. ADAM combina el cálculo de los momentos del gradiente (media y varianza) para cada peso con el descenso de gradiente para actualizar los pesos. El uso de los momentos del gradiente permite que ADAM adapte la tasa de aprendizaje para cada peso de manera individual y, por lo tanto, puede proporcionar una convergencia más rápida y estable en comparación con el descenso de gradiente estándar.

4 Redes neuronales convolucionales

4.1 Introducción

Las redes neuronales convolucionales o, en inglés, convolutional neural networks (CNN) surgieron en la década de 1980, cuando Yann LeCun y otros investigadores las usaron con el objetivo de reconocer caracteres escritos a mano. Sin embargo, su aplicación se limitó debido a la falta de datos y la capacidad de procesamiento de la época.

Con la explosión de datos y la evolución de la tecnología de procesamiento gráfico en la década de 2010, las CNN se convirtieron en una herramienta poderosa en el campo del reconocimiento de imágenes. Desde entonces, se han utilizado en una amplia variedad de aplicaciones, incluyendo la clasificación de imágenes médicas, la detección de objetos en vídeos, la identificación de emociones en el rostro humano y, más recientemente, en la predicción de series temporales.

Una de las razones por las que las CNN son efectivas en la predicción de series temporales es que pueden aprender características de las series de tiempo en diferentes escalas temporales. Al igual que en el procesamiento de imágenes, las capas convolucionales en una CNN para series temporales pueden aprender filtros que se aplican a ventanas de tiempo en la serie. Esto permite a la CNN detectar patrones en diferentes escalas temporales, lo que puede ser especialmente útil en la predicción de series con patrones complejos y no lineales.

4.2 Arquitectura de las redes neuronales convolucionales

En esta sección se abordará una explicación sobre la arquitectura. Para ello, aunque este trabajo está dedicado a la predicción de series temporales, se explicará la arquitectura dedicada a la clasificación de imágenes y se especificarán cuales son las diferencias en cuanto a la aplicación para la predicción de series temporales.

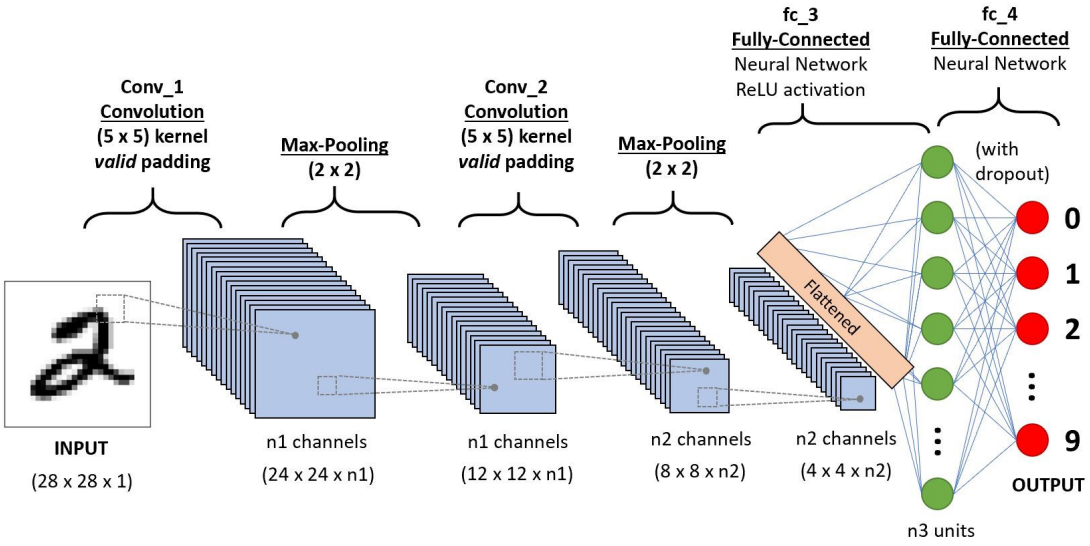


Figura 4.1: Arquitectura de una CNN que clasifica números del 0 al 9

Una arquitectura típica de una CNN consta de varias capas que se encargan de extraer características de las imágenes de entrada y generar una salida final. A continuación, se describen brevemente las principales partes de una CNN:

1. Capa de convolución: Esta capa se encarga de aplicar un conjunto de filtros (kernels) a la imagen de entrada para extraer características relevantes. Los filtros se desplazan por la imagen y se van multiplicando por los píxeles de la misma para obtener una matriz de características. Estas características se pueden interpretar como activaciones de neuronas que se activan cuando se detecta una cierta característica. A esta operación se le conoce como convolución.
2. Capa de pooling: Después de aplicar la capa de convolución, es común aplicar una capa de pooling para reducir la dimensionalidad de las características y

hacer que el modelo sea más eficiente. La capa de pooling toma una ventana de tamaño fijo y calcula una única salida para esa ventana, por ejemplo, tomando el máximo (Max Pooling, como en la figura 4.1) o la media (Average Pooling) de los valores dentro de la ventana. El resultado es una imagen más pequeña, pero con características aún relevantes.

3. Capa de flatten: Esta capa convierte las características 2D (o 3D) de la imagen en un vector 1D, que puede ser procesado por una capa totalmente conectada. Es necesario aplanar la imagen antes de pasarla a la capa fully connected.
4. Capa fully connected: La capa fully connected se encarga de realizar la clasificación final utilizando las características extraídas de la imagen. Esta capa es similar a una red neuronal clásica, donde cada neurona está conectada a todas las neuronas de la capa anterior y cada una tiene un peso y un sesgo que se ajustan durante el entrenamiento. Finalmente, se utiliza una función de activación en la capa de salida para obtener la salida final del modelo.

En cuanto a su aplicación a la predicción de series temporales, la idea es similar pero en lugar de tener una imagen como entrada, se tiene una secuencia temporal. La capa de convolución se utiliza para extraer características relevantes de la secuencia temporal y la capa fully connected se utiliza para realizar la predicción final. Las capas de pooling y flatten pueden ser opcionales dependiendo de la naturaleza de los datos de entrada.

4.2.1 Capa de convolución

La convolución es una operación matemática que se utiliza para combinar dos funciones y producir una tercera función que representa la forma en que una de las funciones "se desliza" sobre la otra y se superpone. En el contexto de las redes neuronales convolucionales (CNN), la convolución se utiliza para extraer características importantes de una imagen o serie temporal de entrada.

Formalmente, la convolución de dos funciones f y g se define como sigue:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau dx$$

Donde $*$ denota la operación de convolución, t es el tiempo y τ es el tiempo desplazado. En otras palabras, la convolución de dos funciones se calcula integrando su

producto después de que una de las funciones se haya desplazado en el tiempo.

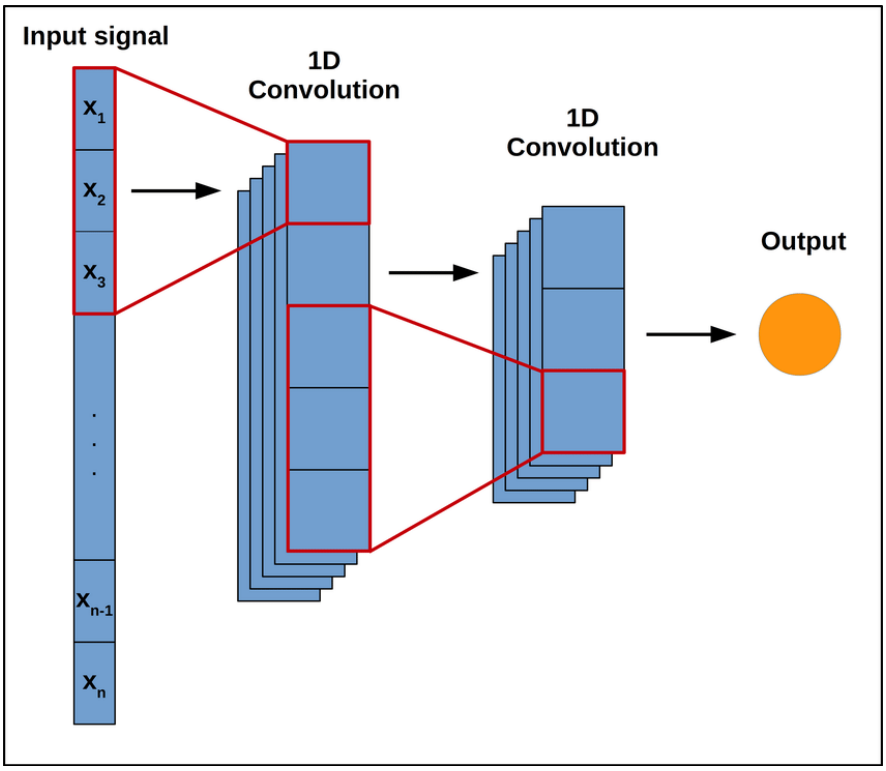


Figura 4.2: Convolución 1D encadenadas para una serie temporal

Sin embargo, nuestros datos de entrada son discretos por lo que se tiene que modificar la expresión anterior. Además, se adaptará según la dimensión de los datos. Así pues, dados el tensor de entrada x y el tensor del kernel h se produce el tensor de salida y:

Convolución 1D:

$$y[t] = \sum_{l} x[t-l]h[l]$$

Convolución 2D:

$$y[i,j] = \sum_{l,m} x[i-l, j-m]h[l,m]$$

Convolución 3D:

$$y[i,j,k] = \sum_{l,m,n} x[i-l, j-m, k-n]h[l,m,n]$$

La convolución 1D se utiliza para datos de series temporales, mientras que la convolución 2D se utiliza para imágenes en blanco y negro y la convolución 3D para imágenes a color. En cada caso, el tensor de entrada x se convoluciona con un tensor de kernel h, que se desliza por el tensor de entrada para producir un tensor de salida y.

En el contexto de una serie temporal multivariante, es común pensar que la convolución se debe aplicar en la dimensión correspondiente al número de variables. No obstante, es posible aplicar la convolución 1D de manera individual en cada variable para obtener un mapa de características con varias capas. De esta forma, se puede explorar la información de manera más detallada y efectiva.

Cabe señalar que en la práctica se sustituye la convolución por la llamada correlación cruzada:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau dx$$

Esto se debe a que la operación de convolución se puede expresar como una correlación cruzada con los filtros invertidos. Sin embargo, la correlación cruzada es más eficiente de calcular, ya que no requiere invertir los filtros. Además, los algoritmos de aprendizaje automático se han desarrollado para la correlación cruzada, por lo que se utiliza en su lugar en la mayoría de las implementaciones de CNN.

Por último, se suele aplicar una capa ReLU a la salida tras la convolución. Esto se hace para distinguir los rasgos en las imágenes, los cuales son altamente no lineales por lo general. No obstante, y más aún para la predicción de series temporales, nada impide usar otra función de activación como la función sigmoide o la tangente hiperbólica.

4.2.2 Capa de pooling

La capa de pooling es una capa común en las CNN y es utilizada para reducir la dimensionalidad de los datos. En lugar de aplicar filtros a cada punto de los datos de entrada, la capa de pooling reduce el tamaño de los datos de entrada aplicando un filtro a cada subregión estos. Esta reducción de la dimensionalidad hace que el modelo sea más rápido y menos propenso a sobreajustar.

La capa de pooling se puede implementar utilizando diferentes operaciones, siendo las más comunes el max-pooling y el average-pooling. El max-pooling toma el valor máximo de la subregión de la entrada que está cubierta por el filtro, mientras que el average-pooling toma el promedio de la subregión de la entrada.

En el caso de series temporales, la capa de pooling se aplica de la siguiente manera: si la entrada es un vector x de longitud N y el tamaño del filtro de pooling es p, la salida es un vector y de longitud $M = \left\lceil \frac{N}{p} \right\rceil$. Para el max-pooling, la operación se define como:

$$y_i = \max_{j=ip}^{(i+1)p-1} x_j, \quad i = 0, \ldots, M-1$$

donde i es la posición de la subregión de la entrada y j es el índice de los elementos dentro de la subregión. Es decir, la operación toma el máximo valor dentro de cada subregión de tamaño p y lo asigna a la posición correspondiente en el vector de salida.

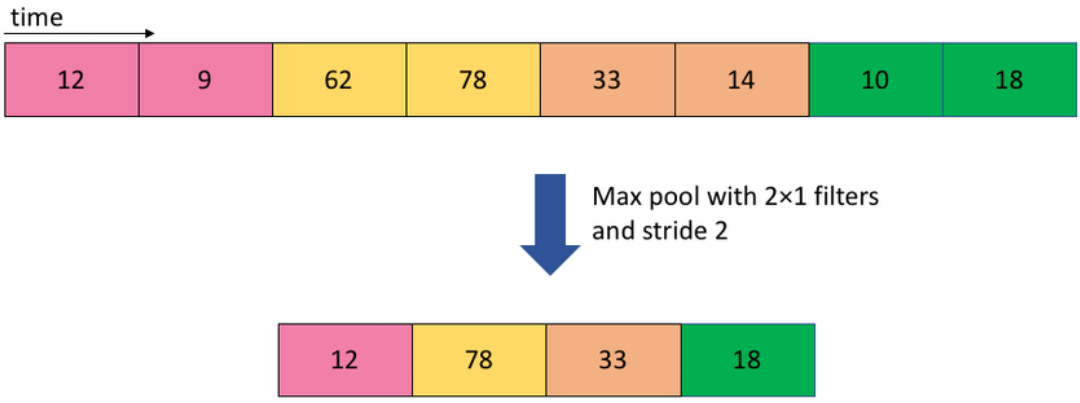


Figura 4.3: Ejemplo de max-pooling para una serie temporal

Para el average-pooling, la operación se define como:

$$y_i = \frac{1}{p} \sum_{j=ip}^{(i+1)p-1} x_j, \quad i = 0, \ldots, M-1$$

es decir, toma el promedio de los valores dentro de cada subregión de tamaño p y lo asigna a la posición correspondiente en el vector de salida.

4.2.3 Capa de flattening

La capa de flattening es una capa que se utiliza en las CNN para convertir la salida de la capa de convolución o pooling en un vector unidimensional que puede ser alimentado a una capa densa.

Si tenemos un tensor de entrada de tamaño $m \times n \times c$, donde m es la longitud de la serie temporal o el ancho de la imagen, n es el número de variables de la serie temporal o la altura de la imagen y c es el número de mapas de características , la capa de flattening simplemente lo convierte en un vector unidimensional de tamaño mnc. En el caso de imágenes 3D se debe añadir una nueva dimensión correspondiente a los 3 canales de colores.

Este vector unidimensional contiene información sobre las características de la serie temporal o la imagen, que se pueden utilizar para predecir los valores futuros de la serie o para la clasificación de la imagen.

4.2.4 Capa fully conected

La capa fully connected (totalmente conectada) en una CNN es una capa de neuronas en la que todas las neuronas de entrada están conectadas con todas las neuronas de salida. Esta capa se utiliza para aprender las características finales de la entrada, y se coloca al final de la arquitectura de la CNN.

En el caso de la predicción de series temporales, la capa fully connected está formada de una o varias capas densas y su salida debe estar compuesta por un número de neuronas igual al número de pasos de tiempo que se deseen predecir. Es decir, si se desea predecir la serie temporal de los próximos 5 pasos de tiempo, se debería tener una capa fully connected con 5 neuronas de salida. Incluso se podría adaptar la red para predecir más de una variable de una serie temporal multivariante al mismo tiempo. De esta manera, la red neuronal aprende a mapear las características extraídas de la serie temporal a las predicciones correspondientes.

En el caso de la clasificación de imágenes, la capa FC se compone de varias capas densas y la salida final de la capa FC tendrá una dimensión igual al número de clases de la clasificación. Por ejemplo, si se tiene un problema de clasificación de imágenes con 10 clases, la capa FC final tendría varias capas densas con activaciones no lineales, como ReLU o tanh, y su salida tendría una dimensión igual a 10, correspondiente al

número de clases de la clasificación. Además, se suele usar la función de softmax:

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Esta función normaliza los valores de salida, de manera que la suma de estas es igual a 1. De esta forma, se pueden interpretar las salidas como probabilidades de pertenencia a cada una de las clases en un problema de clasificación.

5 Redes neuronales recurrentes

5.1 Introducción

Las redes neuronales recurrentes o, en inglés, recurrent neural networks (RNN) son un tipo de arquitectura de red neuronal que se utilizan para procesar datos secuenciales. La idea de utilizar una red neuronal para procesar secuencias se remonta a los años 80, cuando las redes neuronales convencionales se utilizaban para analizar y clasificar imágenes y patrones. Sin embargo, la capacidad de las redes neuronales para procesar secuencias fue limitada en aquel entonces debido a la falta de mecanismos para manejar dependencias temporales a largo plazo.

Las RNN fueron desarrolladas por varios investigadores en el campo de la inteligencia artificial y el aprendizaje automático a lo largo de varias décadas. Sin embargo, la idea de utilizar una red neuronal con conexiones recurrentes se atribuye generalmente a Paul Werbos, quien en 1975 publicó un artículo titulado "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences" en el que propuso la idea de utilizar conexiones recurrentes en una red neuronal para modelar secuencias de datos, como el habla y el lenguaje natural.

En 1982, John Hopfield publicó un artículo sobre una red neuronal con retroalimentación (feedback) que tenía la capacidad de almacenar y recuperar patrones, lo que sentó las bases teóricas de las redes neuronales recurrentes modernas. Finalmente, en la década de 1990, Sepp Hochreiter y Jürgen Schmidhuber propusieron el uso de las redes neuronales recurrentes con una arquitectura conocida como Long Short-Term Memory (LSTM), que resolvió los problemas de gradientes explotantes y desvanecientes que habían limitado la eficacia de las RNN anteriores.

Estos avances permitieron a las redes neuronales procesar secuencias de longitud variable. En lugar de procesar cada entrada de la secuencia de forma independien-

te, las RNN utilizan una memoria interna que les permite capturar las dependencias temporales a largo plazo y adaptarse a diferentes patrones en la secuencia.

En la actualidad, las RNN han encontrado una amplia gama de aplicaciones en diversos campos, desde el procesamiento del lenguaje natural hasta el reconocimiento de voz, la generación de texto, la traducción automática, la clasificación de series temporales y la predicción de valores futuros en datos de tiempo continuo. Además, las RNN han sido utilizadas con éxito en la generación de música y la creación de arte generativo, lo que demuestra su potencial en el campo de la creatividad computacional.

5.2 Arquitectura de las redes neuronales recurrentes

La clave para entender las RNN es que utilizan información de iteraciones anteriores en su cálculo, lo que les permite modelar secuencias de datos. La capa oculta de una RNN tiene conexiones de retroalimentación, lo que significa que los resultados de una iteración se utilizan como entrada para la siguiente iteración.

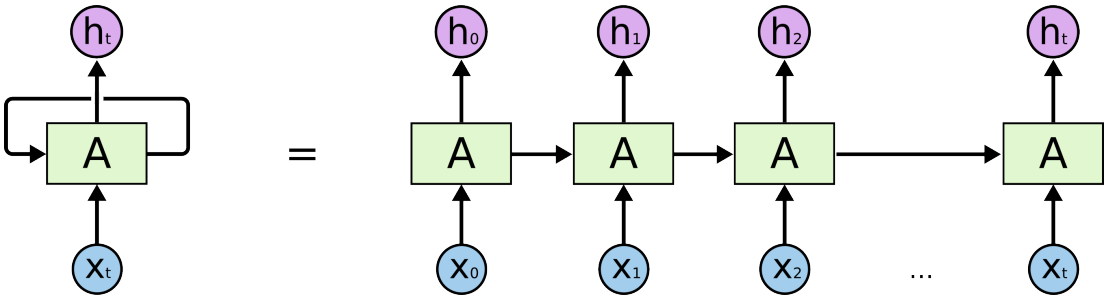


Figura 5.1: Bucle de una RNN

La arquitectura básica de una RNN es un ciclo de retroalimentación de información que permite a la red procesar secuencias de datos de entrada de longitud variable. Los datos de entrada $X = (x_1, x_2, ..., x_T)$ fluyen hacia adelante a través de la red, alimentando la capa oculta A en cada paso de tiempo y produciendo una salida h_t en cada momento t de la secuencia. La salida de la capa oculta puede alimentar la siguiente capa oculta o utilizarse para producir la salida final de la red.

Cabe destacar que los datos de entrada en una red neuronal recurrente pueden ser vectores o incluso matrices, dependiendo del problema en cuestión. En el caso de la

predicción de series temporales multivariantes, los datos de entrada serían matrices que contienen múltiples variables en cada punto de tiempo.

Sean f y g las funciones de activación de la capa oculta y la capa de salida respectivamente, y y_t las salidas de la red para cada instante t. Entonces la salida de una red neuronal recurrente se puede expresar de la siguiente manera:

$$h_t = f(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$
$$y_t = g(W_{yh}h_t + b_y)$$

Donde:

- x_t es la entrada en el tiempo t
- h_t es la salida oculta en el tiempo t
- y_t es la salida final en el tiempo t
- W_{hx} es la matriz de pesos de la capa de entrada a la capa oculta
- W_{hh} es la matriz de pesos de la capa oculta a sí misma
- W_{yh} es la matriz de pesos de la capa oculta a la capa de salida
- b_h es el sesgo de la capa oculta
- b_y es el sesgo de la capa de salida

En el apartado de las redes LSTM se usará una notación más compacta que la anterior. Esto se debe a que, concatenando h_{t-1} con x_t y las matrices de pesos, podemos escribir lo siguiente:

$$W_{hx}x_t + W_{hh}h_{t-1} = W \cdot [h_{t-1}, x_t]$$

Según el patrón de entrada y salida, de la red, las RNN se pueden clasificar en tres tipos principales de arquitecturas: one-to-many, many-to-one y many-to-many.

En una arquitectura one-to-many, la red neuronal recibe un único vector de entrada y genera una secuencia de vectores de salida. Un ejemplo típico de este tipo de red es el modelo de generación de subtítulos de imágenes, donde la red recibe una imagen y genera una secuencia de palabras que describen la imagen. En este caso, la red recibe una única imagen como entrada y produce una secuencia de vectores de salida que corresponden a las palabras del subtítulo.

En una arquitectura many-to-one, la red neuronal recibe una secuencia de vectores de entrada y genera un único vector de salida. Un ejemplo común de este tipo de

red es el análisis de sentimiento, donde la red recibe una secuencia de palabras que forman una oración y produce un único vector de salida que representa el sentimiento expresado en la oración.

Por último, en una arquitectura many-to-many, la red neuronal recibe una secuencia de vectores de entrada y produce una secuencia de vectores de salida. Esta arquitectura se puede dividir en dos subtipos, dependiendo de si la longitud de la secuencia de entrada y la de la secuencia de salida son iguales o diferentes. Si son iguales, se habla de una arquitectura many-to-many del tipo "encoder-decoder". Por ejemplo, la traducción automática es un ejemplo común de este tipo de red, donde la red recibe una secuencia de palabras en un idioma y produce una secuencia de palabras en otro idioma. Si la longitud de la secuencia de entrada y la de la secuencia de salida son diferentes, se habla de una arquitectura many-to-many del tipo "sequence labeling". Un ejemplo típico de este tipo de red es el reconocimiento de habla, donde la red recibe una secuencia de señales de audio y produce una secuencia de fonemas correspondientes.

Incluso podríamos hablar de una arquitectura one to one. Esto puede ser útil en algunas aplicaciones específicas donde la entrada y salida están relacionadas de manera secuencial, pero no hay una secuencia temporal en sí misma. Por ejemplo, una tarea de clasificación de imágenes podría ser abordada utilizando una arquitectura RNN configurada como one to one, donde cada imagen se considera una secuencia de una sola entrada y se produce una única salida de clasificación. Sin embargo, en general, la arquitectura RNN se utiliza más comúnmente para problemas que involucran secuencias temporales.

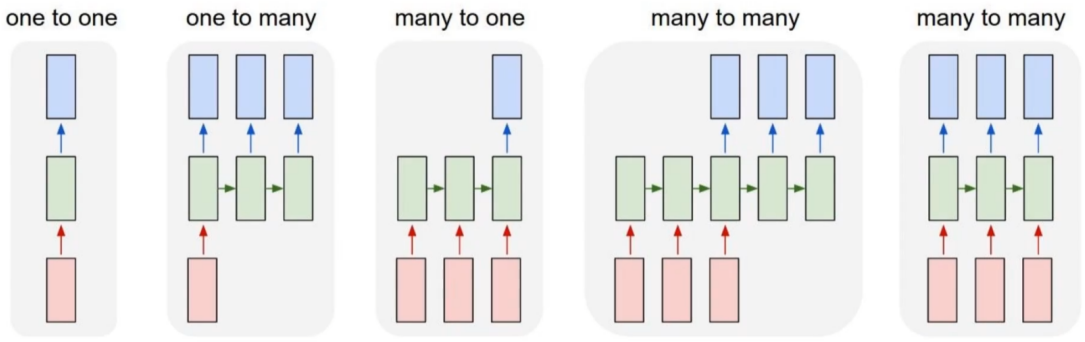


Figura 5.2: Arquitecturas de una RNN según patrón de entrada y salida

En general, cada tipo de arquitectura de RNN es adecuado para diferentes tipos de problemas y, por lo tanto, es importante elegir la arquitectura adecuada para la tarea

específica a resolver.

5.3 Retropropagación a través del tiempo

El algoritmo de retropropagación a través del tiempo (BPTT) es un método utilizado para entrenar las RNN. La idea principal detrás de este algoritmo es aplicar la retropropagación del error en la red, de manera similar a lo que se hace en una red neuronal convencional, pero a través de una secuencia de tiempo.

La retropropagación del error en una RNN se realiza en dos etapas: propagación hacia adelante y propagación hacia atrás. En la propagación hacia adelante, se calculan las salidas h_t y y_t de la red neuronal recurrente, tal como se describió anteriormente. En la propagación hacia atrás, se calculan los gradientes de los parámetros de la red en función del error de salida.

El algoritmo BPTT se puede expresar como:

1. Inicializar los pesos de forma aleatoria o según alguna estrategia predefinida.
2. Para cada instante de tiempo t en la secuencia, calcular las salidas h_t y y_t mediante la propagación hacia adelante de la RNN.
3. Calcular los errores E_t para cada salida de la red con tiempo t. Podríamos expresar el error total como $E = \frac{1}{2}\sum(y_i - y_i^*)^2$, con y_i^* el valor esperado.
4. Calcular los gradientes del error para cada tiempo t con respecto a los parámetros de la red, utilizando la retropropagación del error como se hace en una red neuronal convencional:

$$\frac{\partial E_t}{\partial W_{yh}} = \frac{\partial E_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_{yh}}$$

$$\frac{\partial E_t}{\partial W_{hh}} = \sum_{i=t}^{T} \frac{\partial E_i}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hh}}$$

$$\frac{\partial E_t}{\partial W_{hx}} = \sum_{i=t}^{T} \frac{\partial E_i}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_{hx}}$$

   Donde los sumatorios de las dos últimas expresiones representan la propagación hacia atrás de los gradientes a través del tiempo, es decir, la retropropagación a través del tiempo (BPTT). Es decir, los cálculos se realizan comenzando en T y continuando hacia los valores de t inmediatamente anteriores.

5. Actualizar los parámetros de la red utilizando los gradientes calculados en el paso anterior y un algoritmo de optimización, como el descenso de gradiente.

Este proceso se repite para cada instancia en la secuencia de entrada, y se utiliza para ajustar los pesos de la red de manera iterativa y actualizar su capacidad para hacer predicciones precisas.

A pesar de la potencia del algoritmo, existe el problema de las dependencias a largo plazo, el cuál es un problema fundamental en las RNN. En general, las RNN son muy efectivas para procesar secuencias de datos en las que las dependencias a corto plazo son importantes, pero a menudo tienen dificultades para capturar las dependencias a largo plazo.

El problema de las dependencias a largo plazo es una limitación importante en las redes neuronales recurrentes. Cuando una red tiene que procesar una secuencia de entrada con información que se extiende a lo largo de muchas etapas, el gradiente de error que se retropropaga a través de la red puede disminuir rápidamente a medida que se retrocede en el tiempo, lo que dificulta la actualización de los pesos en las capas más profundas.

El desvanecimiento del gradiente es una de las principales causas de este problema. En una red neuronal recurrente, la actualización de los pesos se basa en el cálculo del gradiente de error con respecto a los pesos de cada capa en cada paso de tiempo. Si el gradiente es pequeño, los pesos no se actualizarán adecuadamente, lo que puede llevar a un rendimiento deficiente de la red. Este problema suele ser especialmente notorio cuando se utilizan funciones de activación que tienen derivadas cercanas a cero, como la función sigmoide, en combinación con una arquitectura de red profunda.

Existen varias técnicas para tratar el problema de las dependencias a largo plazo en las RNN, como la inicialización cuidadosa de los pesos de la red, funciones de activación con derivadas más grandes como la función ReLU, el uso de arquitecturas de redes neuronales más complejas, como las redes neuronales LSTM y GRU, y el uso de técnicas de regularización como el dropout. Estas técnicas pueden ayudar a las RNN a capturar relaciones a largo plazo y mejorar su rendimiento en tareas que implican el procesamiento de secuencias de datos.

5.4 Redes LSTM

Las redes LSTM (Long Short-Term Memory) son un tipo especial de redes neuronales recurrentes diseñadas para solucionar el problema de desvanecimiento del gradiente que ocurre en las RNN con capas recurrentes convencionales.

Las RNN tienen una estructura que se repite en forma de cadena. Estos módulos repetitivos de red neuronal se utilizan para procesar la información secuencial, lo que permite que la red pueda "recordar" información previa y utilizarla para la predicción o clasificación de datos futuros. En una RNN convencional, el módulo de repetición es una estructura simple, como una sola función de activación, que se utiliza para procesar la entrada actual y la salida anterior del modelo. Las redes LSTM tienen esta misma estructura pero con una célula mucho más compleja, véase en la figura 5.3. En este diagrama cada línea lleva datos desde el final de un nodo al inicio de otro. Estos datos se concatenan cuando dos líneas se juntan y se copian cuando las lineas se separan. Los círculos rosas son sumas o productos puntuales y los rectángulos amarillos representan funciones de activación.

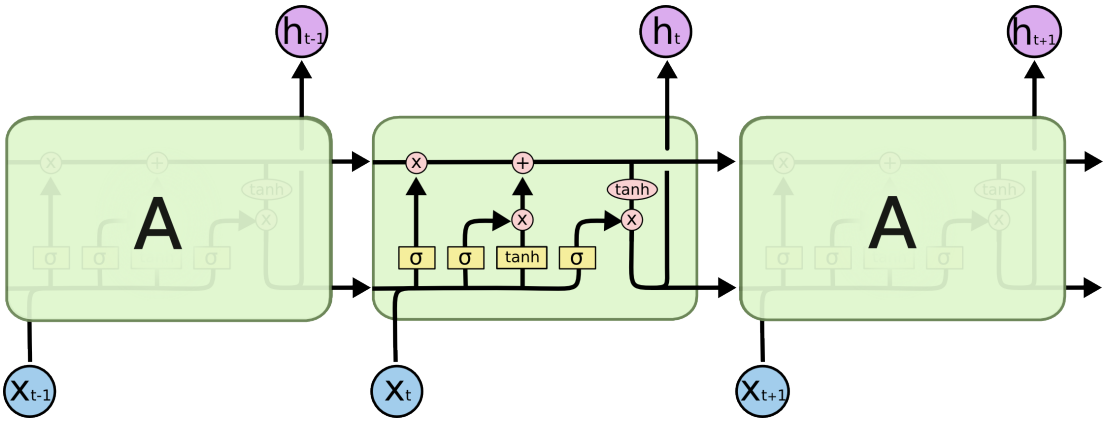


Figura 5.3: Arquitectura LSTM

La clave del LSTM es la celda de memoria C_t que podemos observar en la parte de arriba del diagrama de la figura 5.4 como una línea horizontal. Esta tiene la capacidad de retener información relevante, incluso a largo plazo, de entradas anteriores.

Las redes LSTM tiene la habilidad de agregar o eliminar información a la celda de memoria. Esta habilidad está regulada por estructuras llamadas puertas. Las puertas permiten que la información pase de manera opcional y están compuestas por una capa de red neuronal sigmoide y una operación de producto puntual.

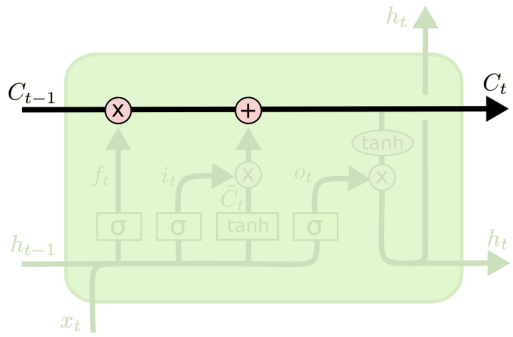
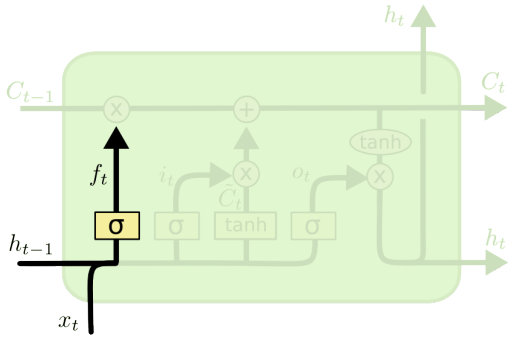


Figura 5.4: Celda de memoria

La capa sigmoide produce valores entre cero y uno, que determinan cuánta información se permitirá pasar. Un valor de cero significa que no se permitirá que pase nada, mientras que un valor de uno significa que se permitirá que pase todo. Una LSTM tiene tres puertas de este tipo, las cuales se utilizan para proteger y controlar la celda de memoria en la red neuronal recurrente. Estas son la forget gate, input gate y output gate.

La forget gate (figura 5.5) decide qué información se va a descartar y qué, por tanto, no pasará a la celda de memoria. Para ello toma el estado oculto anterior h_{t-1} y la entrada actual x_t, que junto a la matriz de pesos W_f y el vector de sesgo b_f pasan por la función de activación sigmoide.



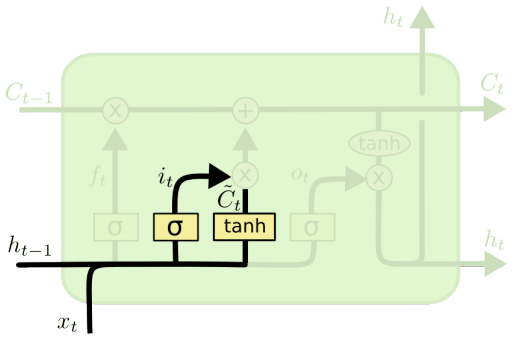
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figura 5.5: Forget gate

Si uno de los valores de f_t es cercano a 0, la red neuronal terminará eliminando esa porción de información, mientras que, si este valor es cercano a 1, la información llegará a la celda de estado.

La input gate (figura 5.6) permite actualizar la memoria. Se toma de igual manera

h_{t-1} y x_t y se pasan junto a W_i y b_i por la función sigmoide. Esto produce como resultado i_t y la intención es preservar los valores cercanos a 1. Por otro lado, tenemos una capa *tanh*, con pesos W_C y sesgos b_c, que crea el candidato $\tilde{C}_t$.

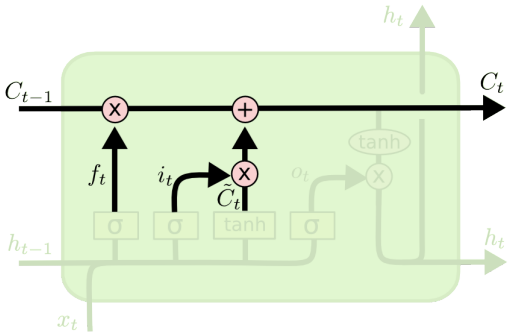


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figura 5.6: Input gate

Gracias a lo explicado anteriormente, se puede pasar a actualizar C_{t-1}. Esto se hará primero realizando $f_t * C_{t-1}$, de tal forma que se olvidará información según lo cercano a 0 que sea cada valor de f_t.

A continuación se tomará $i_t * \tilde{C}_t$, que forman los valores candidatos a añadirse a la memoria, y se sumará. Véase en la figura 5.7.



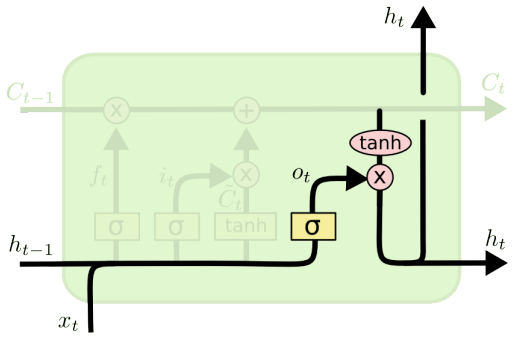
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figura 5.7: Actualización de la memoria

Por último, se deberá calcular el nuevo estado oculto h_t, que se hará con la ayuda de la output gate (figura 5.8). Para ello primero se pasa por la función sigmoide y así poder filtrar que valores de la memoria C_t se conservarán en el nuevo estado oculto. Esto viene acompañado de los pesos W_o y los sesgos b_o. Igualmente, se pasará C_t por la *tanh* para convertirlos en valores entre -1 y 1, el rango que tiene precisamente el

estado oculto. Finalmente multiplicamos estas dos salidas para filtrar los valores en un nuevo estado oculto h_t.

Además de la arquitectura tradicional de LSTM, existen otros modelos de redes neuronales recurrentes que han sido propuestos en la literatura. Algunos de ellos con modificaciones sencillas como el LSTM Peephole. Esta variante introduce conexiones adicionales entre las compuertas y la celda de memoria, permitiendo que la información de la celda de memoria influya directamente en las compuertas.



$$o_t = \sigma\left(W_o\ \left[h_{t-1}, x_t\right]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figura 5.8: Output gate

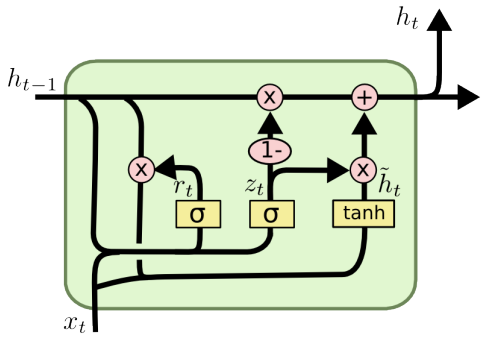
Otras variantes de LSTM difieren bastante más de la arquitectura original. Dos de los modelos más populares son las Gated Recurrent Unit (GRU) y las LSTM convolucionales.

5.5 Redes GRU

Una de las ventajas de una Gated Recurrent Unit (GRU) con respecto a una LSTM es que tiene menos parámetros y por lo tanto requiere menos recursos computacionales para entrenar y ejecutar. Esto se debe a que una GRU utiliza menos compuertas y operaciones que una LSTM, lo que resulta en una red neuronal más simple. Además, en algunos casos, las GRU han demostrado tener un rendimiento comparable o incluso mejor que las LSTM en determinadas tareas, como en el procesamiento del lenguaje natural. Sin embargo, en otras tareas, como en la generación de secuencias de tiempo, las LSTM todavía pueden ser superiores.

Como podemos ver en la figura 5.9, en una GRU también se utilizan compuertas para controlar el flujo de información. Sin embargo, en lugar de tener una compuerta

de entrada y una compuerta de olvido, como en una LSTM, una GRU tiene una única compuerta de actualización z_t que decide cuánta información debe pasar a través de la capa oculta en cada paso de tiempo.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figura 5.9: Arquitectura GRU

Además, en una GRU no hay una memoria a largo plazo separada como en una LSTM, sino que se utiliza una sola capa oculta para realizar las funciones tanto de memoria a corto plazo como de memoria a largo plazo.

6 Caso práctico con Python

6.1 Introducción

Este capítulo estará dedicado a explicar y discutir el código [1] y los resultados del caso prático implementado en Python. El código completo ha sido principalmente desarrollado en Jupyter Notebook, complementado con el software Spyder.

Se ha utilizado la versión de Python 3.9.16 y la biblioteca **Keras** para la creación de los modelos de redes neuronales. Además, se ha hecho uso de las siguientes bibliotecas adicionales:

- **NumPy**: Librería fundamental para el cálculo numérico.
- **Matplotlib, plotly y seaborn**: Para la visualización de datos.
- **Pandas**: Proporciona estructuras de datos y herramientas para el análisis de datos.
- **Statsmodels**: Biblioteca que sirve para realizar modelos estadísticos que ha sido usada para un análisis previo de las series temporales.
- **Scikit-learn**: Librería de aprendizaje automático. Ha sido utilizada para aportar métricas predefinidas y hacer el escalado de los datos.
- **Time**: Módulo estándar de Python que proporciona funciones para trabajar con fechas y tiempos.
- **Psutil**: Para obtener información sobre el sistema operativo y el uso de recursos del sistema, como la CPU, la memoria y los discos.

El notebook completo abarca varios aspectos y contiene una extensa cantidad de información. En este capítulo, se explicarán al menos las partes más relevantes rela-

[1] `https://github.com/javperman/Machine-learning-for-multivariate-time-series`

cionadas con la teoría desarrollada anteriormente. Se proporcionarán explicaciones detalladas para comprender mejor los conceptos clave.

6.2 Descripción y tratamiento de los datos

El conjunto de datos utilizado en este proyecto está compuesto por datos meteorológicos diarios recopilados en la ciudad de Sevilla desde el 1 de enero de 1983 hasta el 1 de enero de 2023. Estos datos han sido obtenidos de la página de la NASA [2] (facilitada a pie de página), una fuente confiable y ampliamente utilizada para recopilar información meteorológica y climática. Los datos fueron guardados en el fichero con nombre *"Datos_Benito_Villamarin.csv"* en honor al estadio.

In [2]:
```
dataset = pd.read_csv("Datos_Benito_Villamarin.csv", sep = ';')
dataset
```

Out[2]:

	YEAR	MO	DY	T2M_MAX	T2M_MIN	PRECTOTCORR	PS
0	1983	1	1	14.24	0.69	0.00	100.87
1	1983	1	2	13.11	-1.12	0.00	101.00
2	1983	1	3	13.94	0.01	0.00	101.14
3	1983	1	4	16.24	0.91	0.00	101.23
4	1983	1	5	17.07	3.06	0.00	101.09
...	...	...	...	...	...	...	...
14606	2022	12	28	19.92	7.19	0.00	100.36
14607	2022	12	29	17.58	7.56	0.03	100.32
14608	2022	12	30	18.35	8.64	0.12	100.39
14609	2022	12	31	17.90	4.55	0.00	100.30
14610	2023	1	1	19.47	7.50	0.85	100.04

Figura 6.1: Leer y visualizar los datos

Los datos de entrada de los modelos serán el mes (`MO`), la temperatura mínima

[2] `https://power.larc.nasa.gov/data-access-viewer/`

(T2M_MIN) y la temperatura máxima (T2M_MAX) diaria en Sevilla. El objetivo es utilizar estos datos históricos para predecir la temperatura máxima del día siguiente. Por supuesto, se tendrán en cuenta las fechas aunque no sirvan para alimentar el modelo. Con la siguiente instrucción quedarán solo las variables de interés:

```
dataset = dataset.iloc[:, [0, 1, 2, 3, 4]]
```

Con el fin de controlar las fechas diarias en una sola variable, se crea la variable DATE de tipo datetime.

```
dataset["DATE"] = pd.to_datetime(dataset["YEAR"].astype(str) + "-" +
                                 dataset["MO"].astype(str) + "-" +
                                 dataset["DY"].astype(str),
                                 format="%Y-%m-%d").dt.date
dataset.insert(0, "DATE", dataset.pop("DATE"))
dataset
```

Out[4]:

	DATE	YEAR	MO	DY	T2M_MAX	T2M_MIN
0	1983-01-01	1983	1	1	14.24	0.69
1	1983-01-02	1983	1	2	13.11	-1.12
2	1983-01-03	1983	1	3	13.94	0.01
3	1983-01-04	1983	1	4	16.24	0.91
4	1983-01-05	1983	1	5	17.07	3.06
...	...	...	...	...	...	...
14606	2022-12-28	2022	12	28	19.92	7.19
14607	2022-12-29	2022	12	29	17.58	7.56
14608	2022-12-30	2022	12	30	18.35	8.64
14609	2022-12-31	2022	12	31	17.90	4.55
14610	2023-01-01	2023	1	1	19.47	7.50

14611 rows × 6 columns

En el notebook se comprueba que no hay problemas con los tipos de los datos ni existen valores nulos. Sin embargo, ya sea con la descripción estadística de los datos o con un boxplot, se observa que hay valores atípicos en las temperaturas.

```
dataset.describe()
```

Out[7]:

	YEAR	MO	DY	T2M_MAX	T2M_MIN
count	14611.000000	14611.000000	14611.000000	14611.000000	14611.000000
mean	2002.501061	6.522552	15.728629	24.897223	11.481236
std	11.544613	3.449006	8.800936	16.996523	15.758233
min	1983.000000	1.000000	1.000000	-999.000000	-999.000000
25%	1992.500000	4.000000	8.000000	17.640000	6.660000
50%	2003.000000	7.000000	16.000000	23.660000	11.150000
75%	2012.500000	10.000000	23.000000	32.680000	17.080000
max	2023.000000	12.000000	31.000000	46.260000	28.270000

Se identifican los mismos 3 valores atípicos en la temperatura máxima y mínima. En ambos casos se decide realizar una interpolación lineal de estos Por ejemplo, con la temperatura máxima:

```
q1 = np.percentile(dataset["T2M_MAX"], 25)
q3 = np.percentile(dataset["T2M_MAX"], 75)
iqr = q3 - q1
lower_limit = q1 - 1.5*iqr
upper_limit = q3 + 1.5*iqr

outliers = dataset[(dataset["T2M_MAX"] < lower_limit) | (dataset["T2M_MAX"] > upper_li
outliers
        DATE            YEAR        MO        DY        T2M_MAX        T2M_MIN
14566       2022-11-18        2022        11        18        -999.0        -999.0
14567       2022-11-19        2022        11        19        -999.0        -999.0
14568       2022-11-20        2022        11        20        -999.0        -999.0
```

```
# INTERPOLAR
first = dataset.iloc[14565, :]['T2M_MAX']
last = dataset.iloc[14569, :]['T2M_MAX']

for i in range(1,4):
    dataset.loc[14565 + i, 'T2M_MAX'] = first + (i/4)*(last - first)
```

6.3 Análisis gráfico previo de la serie temporal

Aunque no es estrictamente necesario para crear un modelo de redes neuronales, se realiza un análisis superficial de la serie temporal con el fin de comprender mejor la naturaleza de los datos. Para ello, se lleva a cabo la descomposición de la serie de la temperatura máxima en sus componentes de tendencia, estacionalidad y ruido, y se grafican estas componentes. El objetivo es obtener información sobre los patrones temporales y las variaciones en los datos.

Es importante destacar que se ha optado por una descomposición aditiva de la serie, ya que los cambios absolutos en la temperatura suelen ser más relevantes que los cambios relativos.

```
import statsmodels.api as sm
decomposition = sm.tsa.seasonal_decompose(dataset['T2M_MAX'],
                                          model='additive',
                                          period=365)
fig, ax = plt.subplots(4, 1, figsize=(12, 8))
decomposition.observed.plot(ax=ax[0], legend=False)
ax[0].set_ylabel('Observed')
decomposition.trend.plot(ax=ax[1], legend=False)
ax[1].set_ylabel('Trend')
decomposition.seasonal.plot(ax=ax[2], legend=False)
ax[2].set_ylabel('Seasonal')
decomposition.resid.plot(ax=ax[3], legend=False)
ax[3].set_ylabel('Residual')
plt.show()
```

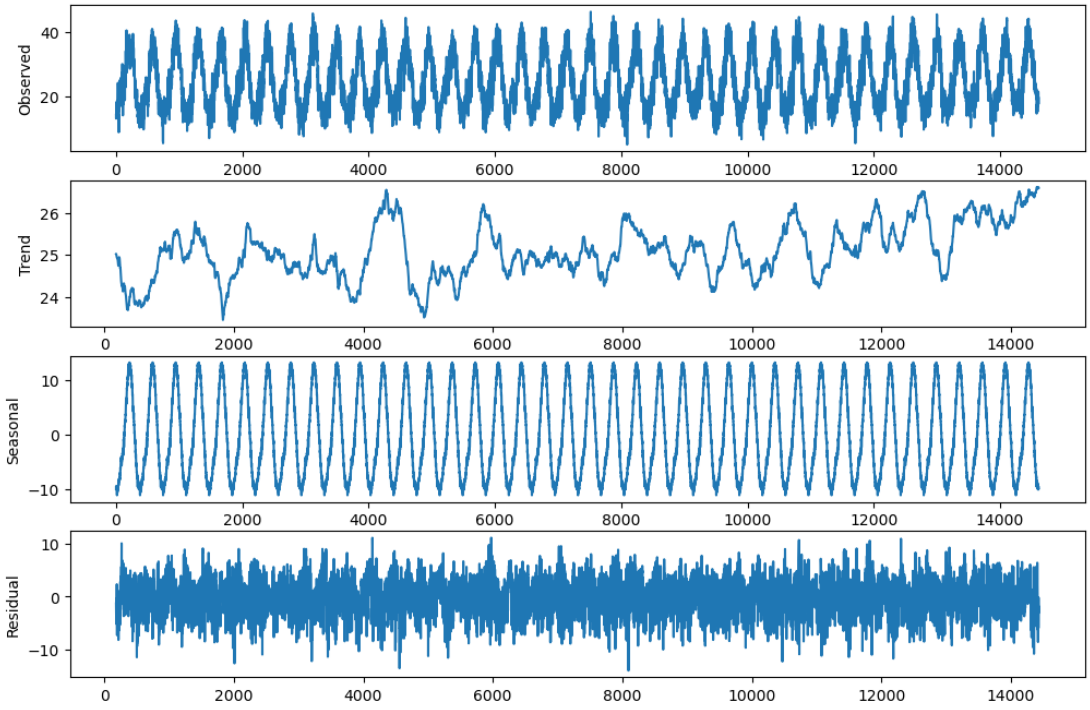


Figura 6.2: Componentes de la temperatura máxima

Para calcular y visualizar la función de autocorrelación (figura 6.3) :

```
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(dataset["T2M_MAX"].values, lags = 365*10)
plt.show()
```

La función sugiere que hay una dependencia o relación entre las temperaturas máximas de los años consecutivos. El hecho de que la amplitud de las ondas disminuya progresivamente indica que la correlación se debilita con el tiempo, es decir, las temperaturas máximas de los años posteriores tienen una relación más débil con las temperaturas máximas de los años anteriores.

Esto podría interpretarse, junto con el hecho de que la tendencia parece ser ligeramente creciente, como un cambio gradual en el patrón climático a lo largo del tiempo. Podría indicar que factores externos, como el cambio climático, están afectando la relación entre las temperaturas máximas anuales.

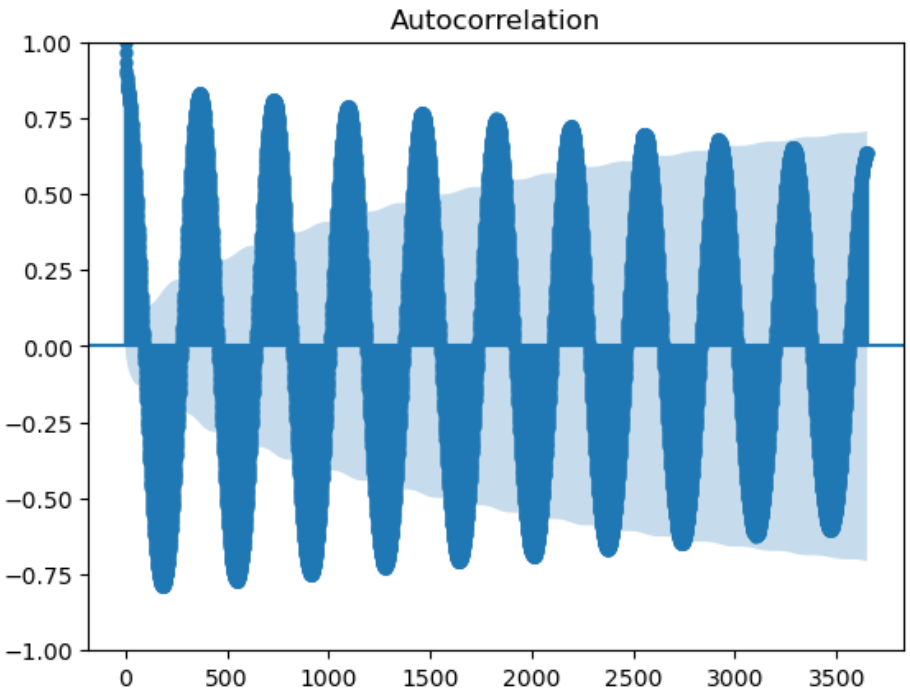


Figura 6.3: Función de autocorrelación de la temperatura máxima

6.4 Preprocesado para los modelos de redes neuronales

Los datos de entrenamiento corresponderán a aquellos tomados hasta el 31 de diciembre de 2019, mientras que los datos de prueba serán los recolectados desde el 1 de enero de 2020 hasta el 1 de enero de 2023.

```
dataset_training = dataset[dataset["YEAR"]< 2020]
dataset_test = dataset[dataset["YEAR"] >= 2020]

# Mes, temp. máxima y temp. mínima de los datos de entrenamiento
training_set = dataset_training.iloc[:, [2,4,5]].values
# Temperatura máxima de los datos de test
real_values = dataset_test.iloc[:, [4]].values
# Fechas de los datos de test, que serán también las de las predicciones
forecast_dates = dataset_test.iloc[:, 0].values
```

Se escalan los datos con el `StandardScaler`, es decir se resta la media de cada característica y se divide por la desviación estándar correspondiente.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler = scaler.fit(training_set)
training_set_scaled = scaler.transform(training_set)
```

El próximo bucle iterará a través de un bucle `for` desde `n_past` hasta la longitud del conjunto de datos de entrenamiento (`training_set_scaled`) menos `n_future + 1`. Esto permite seleccionar ventanas de datos de tamaño `n_past` para el entrenamiento.

```
X_train = []
y_train = []
# En ambas listas vacías se almacenan los datos de entrenamiento
n_future = 1   # Número de días en el futuro que queremos predecir
n_past = 90  # Número de días anteriores que usamos para predecir

for i in range(n_past, len(training_set_scaled) - n_future +1):
    X_train.append(training_set_scaled[i - n_past:i, :])
    y_train.append(training_set_scaled[i + n_future - 1:i + n_future, 1])

X_train, y_train = np.array(X_train), np.array(y_train)
```

Dentro del bucle, se agregan los datos de entrada y de salida correspondientes a las listas `X_train` e `y_train`, respectivamente. `X_train` contiene las secuencias de `n_past` días anteriores como características de entrada para el modelo e `y_train` contiene el valor correspondiente al día siguiente después de las `n_past` observaciones, que es el valor que se desea predecir.

Se repite el proceso para los datos de test con la salvedad de que primero hay que conseguir el set de datos correspondiente. Éste comienza `n_past` días antes del 1 de enero de 2020 (primer día de la predicción para los datos de test).

```
index = dataset_test.index[0]  # Índice del primer registro de 2020
test_set = dataset.iloc[(index-n_past):, [2,4,5]].values
test_set_scaled = scaler.transform(test_set)
X_test = []
y_test = []

for i in range(n_past, len(test_set_scaled) - n_future +1):
    X_test.append(test_set_scaled[i - n_past:i, :])
    y_test.append(test_set_scaled[i + n_future - 1:i + n_future, 1])

X_test, y_test = np.array(X_test), np.array(y_test)
```

6.5 Métricas utilizadas

Con el fin de comparar el rendimiento de los modelos, se usarán las siguientes métricas:

MSE (Error Cuadrático Medio):

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{true}}^{(i)} - y_{\text{pred}}^{(i)})^2$$

MAE (Error Absoluto Medio):

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_{\text{true}}^{(i)} - y_{\text{pred}}^{(i)}|$$

RMSE (Raíz del error cuadrático medio):

$$\text{RMSE} = \sqrt{\text{mean}((y_{\text{pred}} - y_{\text{true}})^2)}$$

MAPE (Error Absoluto Porcentual Medio):

$$\text{MAPE} = \text{mean}(|\frac{y_{\text{true}} - y_{\text{pred}}}{y_{\text{true}}}|) \cdot 100$$

R^2 (R-squared):

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_{\text{true}}^{(i)} - y_{\text{pred}}^{(i)})^2}{\sum_{i=1}^{n}(y_{\text{true}}^{(i)} - \text{mean}(y_{\text{true}}))^2}$$

6.6 Entrenamiento de los modelos

Antes de describir la arquitectura de los modelos, se explicará el proceso de ajuste de sus parámetros. En el notebook se ha implementado una función llamada `train_model`, que tiene como objetivo principal realizar el ajuste del modelo. Además, tiene como objetivo secundario monitorear los recursos del equipo y el tiempo transcurrido durante el proceso. En términos de ajuste del modelo, se proporciona a continuación el código correspondiente, junto con una explicación detallada de cada parte:

```
def train_model(model):
    for epoch in range(epochs_totales):
        history = model.fit(X_train, y_train, epochs=1, batch_size=16,
                            validation_data=(X_test, y_test), verbose=1)
                            # Entrena por una época

        forecast_scaled = model.predict(X_test)
        forecast_scaled_copies = np.repeat(forecast_scaled,
                                           training_set.shape[1], axis=-1)
        forecast = scaler.inverse_transform(forecast_scaled_copies)[:, [1]]

        # Métricas en el conjunto de entrenamiento (escalado)
        losses[epoch] = history.history['loss'][0]
        train_rmses[epoch] = history.history['rmse'][0]
        train_maes[epoch] = history.history['mae'][0]
        train_mapes[epoch] = history.history['mape'][0]
        train_r2s[epoch] = history.history['r2'][0]

        # Métricas en el conjunto de validación (escalado)
        val_losses[epoch] = history.history['val_loss'][0]
        val_rmses[epoch] = history.history['val_rmse'][0]
        val_maes[epoch] = history.history['val_mae'][0]
        val_mapes[epoch] = history.history['val_mape'][0]
```

```
        val_r2s[epoch] = history.history['val_r2'][0]

        # Métricas en el conjunto de validación (sin escalar)
        mse = metrics.mean_squared_error(real_values, forecast)
        rmse2 = rmse(real_values, forecast).numpy()
        mae = metrics.mean_absolute_error(real_values, forecast)
        mape = metrics.mean_absolute_percentage_error(real_values, forecast)
        r2 = metrics.r2_score(real_values, forecast)

        mses[epoch] = mse
        rmses[epoch] = rmse2
        maes[epoch] = mae
        mapes[epoch] = mape
        r2s[epoch] = r2

        # Imprimimos las métricas
        print("Epoch /".format(epoch+1, epochs_totales))
        print("MSE:", mse)
        print("RMSE:", rmse2)
        print("MAE:", mae)
        print("MAPE:", mape)
        print("R2:", r2)
        print()

    return history
```

1. Itera sobre el número total de épocas definidas en `epochs_totales`, que en este caso será siempre 30.
2. Utiliza el método `fit` para entrenar el modelo durante una sola época. Los datos de entrenamiento y las etiquetas de entrenamiento se pasan como argumentos, así como los datos de validación y las etiquetas de validación para monitorear el rendimiento del modelo durante el entrenamiento. El `batch_size=16` indica que las muestras serán propagadas y procesadas en lotes de 16 antes de que se realice una actualización de los pesos del modelo.
3. Después de la época de entrenamiento, se utiliza el modelo para hacer predicciones en el conjunto de pruebas (`X_test`), y se escala inversamente las predicciones utilizando `scaler.inverse_transform`. Las predicciones escaladas se almacenan en la variable `forecast`.
4. Se calculan diferentes métricas de evaluación tanto en el conjunto de entrena-

miento como en el conjunto de validación, y se almacenan en diferentes variables, como `losses`, `train_rmses`, `val_maes`, etc. Estas métricas se obtienen a partir del historial de entrenamiento (`history`) proporcionado por el método `fit`.

5. Se calculan métricas adicionales en el conjunto de validación sin escalar, utilizando funciones de la librería `metrics` de scikit-learn. Estas métricas incluyen el error cuadrático medio (MSE), la raíz cuadrada del error cuadrático medio (RMSE), el error absoluto medio (MAE), el error porcentual absoluto medio (MAPE) y el coeficiente de determinación (R2).
6. Se imprimen las métricas calculadas para cada época.
7. La función devuelve el historial de entrenamiento (`history`).

En las próximas secciones se explicarán las arquitecturas de todos los modelos implementados, explicando el código detalladamente.

6.7 Modelo CNN

El siguiente código crea y compila un modelo de red neuronal convolucional (CNN) con la métrica mse (mean squared error) para la función de pérdida:

```
model_CNN = Sequential()
model_CNN.add(Conv1D(64, kernel_size=2, activation='relu',
                     input_shape=(X_train.shape[1], X_train.shape[2])))
model_CNN.add(Flatten())
model_CNN.add(Dropout(0.2))
model_CNN.add(Dense(32, activation='relu'))
model_CNN.add(Dropout(0.2))
model_CNN.add(Dense(y_train.shape[1]))

model_CNN.compile(optimizer='adam', loss='mse',
                  metrics=[rmse, 'mae', mape, r2])
model_CNN.summary()
```

```
Model: "sequential"
_________________________________________________________________
```

```
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d (Conv1D)             (None, 89, 64)            448

 flatten (Flatten)           (None, 5696)              0

 dropout (Dropout)           (None, 5696)              0

 dense (Dense)               (None, 32)                182304

 dropout_1 (Dropout)         (None, 32)                0

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 182,785
Trainable params: 182,785
Non-trainable params: 0
_________________________________________________________________
```

Este ha sido el modelo con más rapidez para entrenar. La función de pérdida será siempre el mse, tanto para este modelo como para el resto. A continuación, se explica qué hace cada parte del código:

1. Se crea un modelo de tipo secuencial (`Sequential`), que es una pila lineal de capas.
2. Se agregan capas al modelo:
   - Capa convolucional (`Conv1D`) con 64 filtros, un tamaño de kernel de 2 y función de activación 'relu'. La capa recibe una entrada con forma (`X_train.shape[1]`, `X_train.shape[2]`), lo que significa que espera secuencias de datos con `X_train.shape[1]` pasos de tiempo y `X_train.shape[2]` características en cada paso de tiempo.
   - Capa de aplanamiento (`Flatten`) para convertir el tensor de salida de la capa convolucional en un vector unidimensional.
   - Capa de desactivación aleatoria (`Dropout`) con una tasa de desactivación del 20 % para evitar el sobreajuste.
   - Capa densa (`Dense`) con 32 unidades y función de activación 'relu'.

   - Otra capa de desactivación aleatoria (`Dropout`) con una tasa de desactivación del 20 %.
   - Capa densa (`Dense`) con un número de unidades igual a la dimensión de la variable de salida `y_train`.
3. Se compila el modelo utilizando el optimizador "adam" y la pérdida "mse" (error cuadrático medio) como función de pérdida. Además, se especifican métricas adicionales a evaluar durante el entrenamiento, incluyendo "rmse" (raíz cuadrada del error cuadrático medio), "mae" (error absoluto medio), "mape"(error porcentual absoluto medio) y "r2" (coeficiente de determinación).
4. Se imprime un resumen del modelo utilizando el método `summary()` para mostrar la arquitectura y el número de parámetros entrenables.

La capa de Dropout implementa una técnica de regularización en la red neuronal. Durante el entrenamiento, el Dropout aleatoriamente "apaga" (pone a cero) un porcentaje de las salidas de la capa anterior. Esto ayuda a prevenir el sobreajuste al forzar a la red a aprender características más robustas y evitar la dependencia excesiva de ciertas neuronas. Esta técnica se usará también en el resto de modelos.

La parte final del historial tiene la información de las métricas finales, junto a la información sobre el monitoreo del tiempo y los recursos:

```
839/839 [==============================] - 5s 6ms/step - loss: 0.0909 -
rmse: 0.2955 - mae: 0.2367 - mape: 170.1157 - r2: 0.8981 - val_loss: 0.0727 -
val_rmse: 0.2638 - val_mae: 0.2123 - val_mape: 80.4625 - val_r2: 0.0078
35/35 [==============================] - 0s 3ms/step
Epoch 30/30
MSE: 5.360318959195742
RMSE: 2.3152362642278526
MAE: 1.8229395554993735
MAPE: 0.07690912532039344
R2: 0.9236630993239426

Uso de CPU antes del entrenamiento: 25.5 %
Uso de memoria antes del entrenamiento: 80.7 %
Uso de CPU después del entrenamiento: 53.1 %
Uso de memoria después del entrenamiento: 81.5 %
Tiempo total de ejecución: 158 segundos
```

Las métricas en minúscula son las calculadas con los datos sin escalar, tal y como se lanzan al modelo. Las métricas en mayúscula están calculadas en el conjunto de validación sin escalar, con el fin de ser interpretables en los datos reales.

En el notebook se pueden observar los gráficos que muestran el comportamiento de las métricas a lo largo de las épocas. En todos los modelos, se puede apreciar que las métricas mejoran a medida que avanzan las épocas de entrenamiento, o al menos se estabilizan en algún punto.

Además, en cada uno de los modelos se incluye un gráfico de dispersión que muestra la línea de referencia f_t (predicciones) = y_t (valores reales), lo cual permite comparar los valores pronosticados con los reales.

Por último, en el notebook se encuentra la función "generate_plot(forecast)" que crea un gráfico interactivo de la serie temporal junto con la predicción. Sin embargo, es importante mencionar que esta visualización solo puede ser apreciada al ejecutar el código, ya que no se guarda directamente en el notebook.

6.8 Modelo LSTM

```
model_LSTM = Sequential()
model_LSTM.add(LSTM(32, activation='tanh',
               input_shape=(X_train.shape[1], X_train.shape[2]),
               return_sequences = False))
model_LSTM.add(Dropout(0.1))
model_LSTM.add(Dense(y_train.shape[1]))

model_LSTM.compile(optimizer='adam', loss='mse', metrics=[rmse, 'mae', mape, r2])
model_LSTM.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 32)                4608
```

```
 dropout_2 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 4,641
Trainable params: 4,641
Non-trainable params: 0
_________________________________________________________________
```

Este es un modelo simple LSTM. Contiene una capa LSTM con 32 unidades y activación "tanh". La capa recibe una entrada con forma (`X_train.shape[1]`, `X_train.shape[2]`). Al establecer `return_sequences=False`, la capa LSTM devuelve solo la última salida de la secuencia en lugar de una secuencia completa. También se agrega una capa de dropout con una tasa de 0.1.

Parte final del historial:

```
839/839 [==============================] - 16s 19ms/step - loss: 0.0664 -
rmse: 0.2525 - mae: 0.1998 - mape: 163.6501 - r2: 0.9242 - val_loss: 0.0615 -
val_rmse: 0.2406 - val_mae: 0.1946 - val_mape: 59.9417 - val_r2: 0.1983
35/35 [==============================] - 0s 8ms/step
Epoch 30/30
MSE: 4.536952345094846
RMSE: 2.130012287545508
MAE: 1.6710700142372708
MAPE: 0.06949054619467222
R2: 0.9353887551886512

Uso de CPU antes del entrenamiento: 15.4 %
Uso de memoria antes del entrenamiento: 81.8 %
Uso de CPU después del entrenamiento: 39.3 %
Uso de memoria después del entrenamiento: 80.8 %
Tiempo total de ejecución: 489 segundos
```

6.9 LSTM_large

```
model_LSTM_large = Sequential()
model_LSTM_large.add(LSTM(64, activation='tanh',
                     input_shape=(X_train.shape[1], X_train.shape[2]),
                     return_sequences = True))
model_LSTM_large.add(Dropout(0.2))
model_LSTM_large.add(LSTM(64, activation = 'tanh', return_sequences = True))
model_LSTM_large.add(Dropout(0.2))
model_LSTM_large.add(LSTM(32, activation = 'tanh', return_sequences = False))
model_LSTM_large.add(Dropout(0.2))
model_LSTM_large.add(Dense(y_train.shape[1]))

model_LSTM_large.compile(optimizer='adam', loss='mse', metrics=[rmse, 'mae', mape, r2])
model_LSTM_large.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm_1 (LSTM)                (None, 90, 64)            17408

 dropout_3 (Dropout)          (None, 90, 64)            0

 lstm_2 (LSTM)                (None, 90, 64)            33024

 dropout_4 (Dropout)          (None, 90, 64)            0

 lstm_3 (LSTM)                (None, 32)                12416

 dropout_5 (Dropout)          (None, 32)                0

 dense_3 (Dense)              (None, 1)                 33

=================================================================
Total params: 62,881
Trainable params: 62,881
```

```
Non-trainable params: 0
_________________________________________________________________
```

Este modelo tiene múltiples capas y es más complejo que el anterior:

1. Se agrega una capa LSTM al modelo con 64 unidades y activación "tanh". La capa recibe una entrada con forma (`X_train.shape[1]`, `X_train.shape[2]`). Además, se establece `return_sequences=True` para que la capa LSTM devuelva secuencias de salidas en lugar de una única salida.
2. Se agrega una capa de dropout con una tasa de 0.2 para regularizar la red y evitar el sobreajuste.
3. Se agrega otra capa LSTM al modelo con 64 unidades y activación "tanh". También se establece `return_sequences=True` para mantener la salida como una secuencia.
4. Se agrega otra capa de dropout con una tasa de 0.2.
5. Se agrega una tercera capa LSTM al modelo con 32 unidades y activación "tanh". Esta vez, `return_sequences` se establece en `False` para que la capa LSTM devuelva solo la última salida de la secuencia.
6. Se agrega otra capa de dropout con una tasa de 0.2.

A base de prueba y error, se decidió que las tasas de las capas de dropout sean mayores ya que, de ser más bajas, existiría sobreajuste en el modelo.

Parte final del historial:

```
839/839 [==============================] - 48s 57ms/step - loss: 0.0735 -
rmse: 0.2658 - mae: 0.2114 - mape: 147.5932 - r2: 0.9150 - val_loss: 0.0634 -
val_rmse: 0.2463 - val_mae: 0.2015 - val_mape: 58.9298 - val_r2: 0.1256
35/35 [==============================] - 1s 23ms/step
Epoch 30/30
MSE: 4.67338900891665
RMSE: 2.1618022594392508
MAE: 1.7303019704966514
MAPE: 0.07244838465305659
R2: 0.9334457454285938

Uso de CPU antes del entrenamiento: 20.2 %
```

```
Uso de memoria antes del entrenamiento: 81.2 %
Uso de CPU después del entrenamiento: 49.0 %
Uso de memoria después del entrenamiento: 81.5 %
Tiempo total de ejecución: 1565 segundos
```

6.10 Modelo GRU

```
model_GRU = Sequential()
model_GRU.add(GRU(32, activation='tanh',
              input_shape=(X_train.shape[1], X_train.shape[2]),
              return_sequences = False))
model_GRU.add(Dropout(0.1))
model_GRU.add(Dense(y_train.shape[1]))

model_GRU.compile(optimizer='adam', loss='mse', metrics=[rmse, 'mae', mape, r2])
model_GRU.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 32)                3552

 dropout_6 (Dropout)         (None, 32)                0

 dense_4 (Dense)             (None, 1)                 33

=================================================================
Total params: 3,585
Trainable params: 3,585
Non-trainable params: 0
_________________________________________________________________
```

Este es el modelo con menos parámetros y es la misma versión que el Modelo LSTM pero con capas GRU.

Información final del historial:

```
839/839 [==============================] - 14s 16ms/step - loss: 0.0661 -
rmse: 0.2518 - mae: 0.1995 - mape: 156.6762 - r2: 0.9243 - val_loss: 0.0590 -
val_rmse: 0.2350 - val_mae: 0.1871 - val_mape: 59.9105 - val_r2: 0.2532
35/35 [==============================] - 0s 8ms/step
Epoch 30/30
MSE: 4.3519579325900315
RMSE: 2.0861346870684145
MAE: 1.607064167047047
MAPE: 0.06878496803859539
R2: 0.9380232812682567

Uso de CPU antes del entrenamiento: 22.7 %
Uso de memoria antes del entrenamiento: 82.0 %
Uso de CPU después del entrenamiento: 50.1 %
Uso de memoria después del entrenamiento: 80.4 %
Tiempo total de ejecución: 413 segundos
```

6.11 Modelo GRU_large

```
model_GRU_large = Sequential()
model_GRU_large.add(GRU(64, activation='tanh',
                    input_shape=(X_train.shape[1], X_train.shape[2]),
                    return_sequences = True))
model_GRU_large.add(Dropout(0.2))
model_GRU_large.add(GRU(64, activation = 'tanh', return_sequences = True))
model_GRU_large.add(Dropout(0.2))
model_GRU_large.add(GRU(32, activation = 'tanh', return_sequences = False))
model_GRU_large.add(Dropout(0.2))
model_GRU_large.add(Dense(y_train.shape[1]))

model_GRU_large.compile(optimizer='adam', loss='mse', metrics=[rmse, 'mae', mape, r2])
model_GRU_large.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru_1 (GRU)                 (None, 90, 64)            13248

 dropout_7 (Dropout)         (None, 90, 64)            0

 gru_2 (GRU)                 (None, 90, 64)            24960

 dropout_8 (Dropout)         (None, 90, 64)            0

 gru_3 (GRU)                 (None, 32)                9408

 dropout_9 (Dropout)         (None, 32)                0

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 47,649
Trainable params: 47,649
Non-trainable params: 0
_________________________________________________________________
```

Esta es la versión compleja de los GRU. Tiene la misma estructura que el Modelo LSTM_large salvo por usar capas GRU.

Última parte del historial:

```
839/839 [==============================] - 46s 55ms/step - loss: 0.0729 -
rmse: 0.2648 - mae: 0.2112 - mape: 170.6892 - r2: 0.9170 - val_loss: 0.0617 -
val_rmse: 0.2410 - val_mae: 0.1954 - val_mape: 59.4554 - val_r2: 0.2038
35/35 [==============================] - 1s 23ms/step
Epoch 30/30
MSE: 4.547280787445288
RMSE: 2.132435412256439
MAE: 1.6776178343293877
MAPE: 0.07066136227320081
```

```
R2: 0.9352416666881633

Uso de CPU antes del entrenamiento: 25.9 %
Uso de memoria antes del entrenamiento: 80.9 %
Uso de CPU después del entrenamiento: 56.4 %
Uso de memoria después del entrenamiento: 75.8 %
Tiempo total de ejecución: 1433 segundos
```

6.12 Modelo Naive

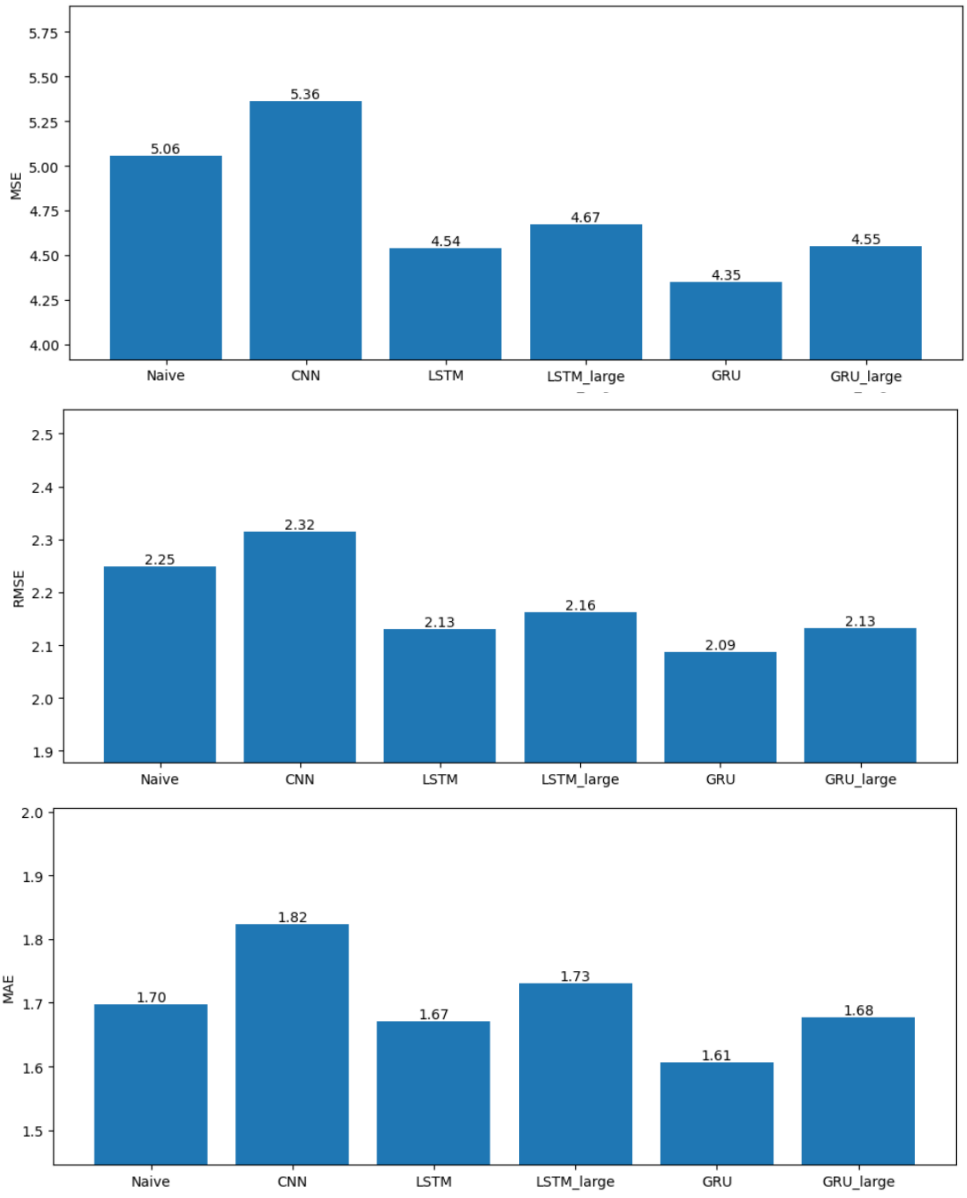
```
# Índice del primer registro de 2020
index = dataset_test.index[0]
# Temperatura máxima desde el 31 de diciembre de 2019
X_train_naive = dataset.iloc[(index-1):, [4]].values

forecast_naive = X_train_naive[:-1] # Simplemente se quita el último valor
```

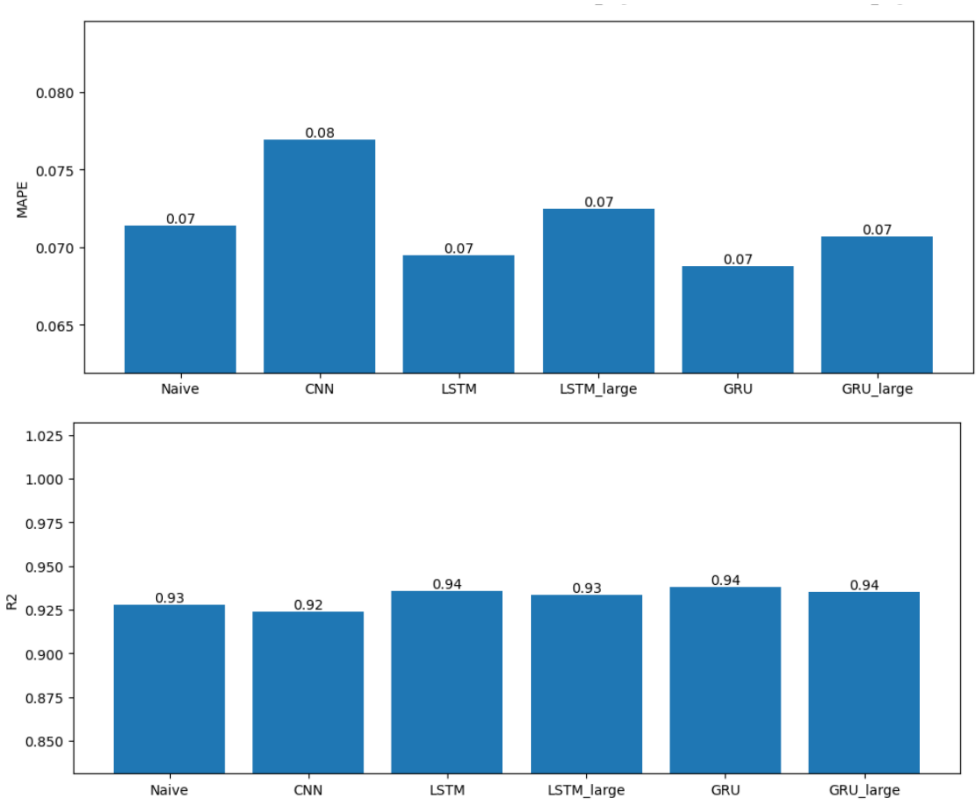
Este último modelo es un "modelo ingenuo" que se basa en la predicción de la temperatura utilizando el valor del día anterior. Se utilizará como punto de referencia para compararlo con los demás modelos. Sus métricas calculadas en los datos de validación sin escalar son:

```
MSE: 5.0570462169553325
RMSE: 2.2487877216303302
MAE: 1.697447584320875
MAPE: 0.0714093714460413
R2: 0.9279820403008495
```

6.13 Discusión de resultados y conclusiones

Estas son las métricas en el conjunto de validación original:





Al observar las métricas de los diferentes modelos, podemos notar que el modelo CNN tiene métricas ligeramente inferiores en comparación con los otros modelos (LSTM, LSTM_large, GRU, GRU_large) y el modelo ingenuo. Existen varias razones posibles por las cuales el modelo CNN puede mostrar un rendimiento inferior en este caso, pero la más clara es que la arquitectura de una CNN está diseñada principalmente para capturar patrones espaciales en datos bidimensionales, como imágenes, donde los filtros convolucionales pueden extraer características visuales relevantes.

En el caso de la predicción de series temporales univariantes, como la temperatura máxima diaria, la estructura espacial no es tan relevante. Sin embargo, podría tener mucho sentido utilizar la arquitectura CNN en el contexto de la predicción de varios días seguidos, en lugar de un solo día. En este escenario, se podrían considerar secuencias de datos más largas, lo que permitiría a la CNN aprender patrones y relaciones más complejas en los datos, capturando así la dependencia temporal a lo largo de varios días. Esto podría mejorar el rendimiento relativo de un modelo CNN con respecto a los demás y aprovechar al máximo su capacidad para extraer características y patrones en datos secuenciales.

Por lo tanto, aunque en este caso específico el rendimiento del modelo CNN fue ligeramente inferior, no se descarta su utilidad en la predicción de series temporales si se ajusta adecuadamente y se adapta a contextos específicos donde se consideren secuencias más largas de datos.

Por otro lado, el modelo LSTM_large muestra, en general, un desempeño ligeramente superior al del modelo ingenuo. Sin embargo, es importante tener en cuenta que el modelo ingenuo presenta un mejor rendimiento en las métricas MAE y MAPE y que el modelo LSTM_large requiere un tiempo considerable para su entrenamiento con nuevos datos. Además, el modelo GRU_large sale favorecido en todas las métricas con respecto a los dos anteriores modelos.

Por último, se observa que los modelos de redes neuronales más simples, LSTM y GRU, muestran un rendimiento destacado en comparación con los demás modelos. Además, el modelo GRU presenta un rendimiento considerablemente superior al LSTM y también se entrena más rápidamente.

6.14 Líneas futuras de investigación

Basado en las métricas y el rendimiento de los modelos desarrollados para la predicción de la temperatura máxima diaria, se pueden identificar varias líneas futuras de investigación:

- **Mejorar la precisión de los modelos existentes**: Aunque los modelos LSTM y GRU han mostrado buenos resultados, aún hay margen para mejorar su rendimiento. Se pueden explorar diferentes arquitecturas de redes neuronales, ajustar los hiperparámetros o aplicar técnicas de regularización para obtener modelos más precisos. Para ajustar los hiperparámetros podrían usarse técnicas como el Grid Search o Random Search.
- **Incorporación de información adicional**: Se puede considerar la inclusión de otros datos relacionados, como datos climáticos adicionales, datos geoespaciales o datos de eventos especiales, para mejorar la precisión de las predicciones. Estos datos pueden ser útiles para capturar patrones o influencias adicionales en la temperatura máxima diaria.
- **Entrenar y evaluar modelos en diferentes ubicaciones geográficas**: Los modelos desarrollados hasta ahora se basan en datos de la ciudad de Sevilla. Sería interesante evaluar el rendimiento de los modelos en otras ubicaciones

geográficas con diferentes características climáticas. Esto ayudaría a determinar la generalización y aplicabilidad de los modelos en diferentes contextos.

- **Investigar técnicas de series temporales más avanzadas**: Aunque los modelos LSTM y GRU son efectivos para modelar datos de series temporales, existen técnicas más avanzadas, como CRNN (convolutional recurrent neural network) o los modelos Transformer, que han demostrado ser exitosos en otros dominios. Incluso existen modelos estadísticos como ARIMA (Autoregressive Integrated Moving Average) o SARIMA (Seasonal Autoregressive Integrated Moving Average), que son ampliamente utilizados en el análisis de series temporales.
- **Extender la predicción de la temperatura máxima diaria a la predicción de varios días consecutivos**. En lugar de limitarse a predecir únicamente un día, se puede explorar la capacidad de los modelos para hacer pronósticos a corto plazo en un horizonte temporal más amplio, como predecir la temperatura máxima de los próximos 3, 5 o incluso 7 días. Como ya se mencionó anteriormente, las CNN podrían jugar un papel importante en este aspecto.

Bibliografía

[1] Hyndman, R. J. y Athanasopoulos, G. (2018). *Forecasting: Principles and Practice* Recuperado de https://otexts.com/fpp2/

[2] Machine Learning Plus. (2019). *Time Series Analysis in Python with Statsmodels.* Recuperado de https://www.machinelearningplus.com/time-series/time-series-analysis-python/

[3] SuperDataScience Team. (2018). *The Ultimate Guide to Artificial Neural Networks (ANN).* Recuperado de https://www.superdatascience.com/blogs/the-ultimate-guide-to-artificial-neural-networks-ann

[4] José David Villanueva García. (2020). *Redes Neuronales desde Cero I: Introducción.* Recuperado de https://www.iartificial.net/redes-neuronales-desde-cero-i-introduccion

[5] McCulloch, W. S. y Pitts, W. (1943). *A Logical Calculus of Ideas Immanent in Nervous Activity.*

[6] Rosenblatt, F. (1958). *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.*

[7] Wilson, A. G., Roelofs, R., Stern, M., Srebro, N., y Recht, B. (2017). *The Marginal Value of Adaptive Gradient Methods in Machine Learning.*

[8] MathWorks. (s.f.). *Convolución.* Recuperado de https://la.mathworks.com/discovery/convolution.html

[9] SuperDataScience Team. (2018). *The Ultimate Guide to Convolutional Neural Networks (CNN).* Recuperado de https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn

[10] Yann LeCun (2020). *PyTorch Deep Learning - Semana 3*. Recuperado de https://atcold.github.io/pytorch-Deep-Learning/es/week03/03-1/

[11] Wu, J. (2017). *Introduction to Convolutional Neural Networks*

[12] Aprende Machine Learning, Na8. (2018). *Cómo funcionan las Convolutional Neural Networks (Visión por Ordenador)*. Recuperado de https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

[13] Hochreiter, S. y Schmidhuber, J. (1997). *Long short-term memory*.

[14] Bengio, Y.; Frasconi, P.; Simard, P. (1993). *The problem of learning long-term dependencies in recurrent networks*. IEEE

[15] Colah. (2015). *Understanding LSTM Networks*. Recuperado de https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[16] Afshine Amidi; Shervine Amidi (2019). *Cheatsheet: Recurrent Neural Networks*. Recuperado de https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

[17] Aditi Mittal (2019). *Understanding RNN and LSTM*. Recuperado de https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e