2023

Regresión en Series Temporales usando Aprendizaje profundo

TRABAJO FIN DE GRADO

MARCOS COUROS GARCÍA

Contenido

Introducción

El objetivo de este trabajo fin de grado (TFG) consiste en evaluar la predicción de series temporales mediante distintas técnicas basadas en aprendizaje profundo o Deep Learning ,y realizar una comparación con modelos clásicos como AR, MA, ARMA, ARIMA o SARIMA y con modelos basados en aprendizaje automático[1].

Desde que se presentó el anteproyecto de este TFG han irrumpido con fuerza dentro del Aprendizaje Profundo los Modelos Largos de Lenguaje (LLM en inglés) como chatGPT y variantes. Inicialmente no se contemplaba explorar el uso de modelos basados en la generación de lenguaje para la predicción de series temporales, pero dada la actualidad de este tipo de modelos haremos también una exploración inicial para ver el comportamiento de modelos de uso general, estando la creación de modelos específicos de LLM fuera del alcance de este TFG entre otras cosas debido al alto coste computacional que requiere este tipo de modelos. Un uso específico de modelos basados en GPT al sector financiero podemos encontrarlo en empresas como Bloomberg, con BloombergGPT [2]

Introducing BloombergGPT, Bloomberg's 50-billion parameter large language model, purpose-built from scratch for finance

March 30, 2023

BloombergGPT outperforms similarly-sized open models on financial NLP tasks by significant margins – without sacrificing performance on general LLM benchmarks

[1] https://epub.ub.uni-muenchen.de/25580/1/MA_Pritzsche.pdf
[2] https://www.bloomberg.com/company/press/bloomberggpt-50-billion-parameter-llm-tuned-finance/

Modelos clásicos

Empezaremos realizando un primer análisis exploratorio de los modelos clásicos exponiendo la base fundamental que los diferencia. Posteriormente haremos una comparación de estos modelos clásicos con los modelos basados en redes neuronales.

Los modelos clásicos han constituido la piedra angular en el análisis de series temporales durante décadas y proporcionan una base sólida sobre la cual se han construido técnicas más avanzadas. Con el objeto de tener una visión de conjunto donde se destacan las principales características de los modelos clásicos presentamos una tabla resumen de los modelos AR (Autorregresivo), MA (Media Móvil), ARMA (Autorregresivo-Media Móvil), ARIMA (Autorregresivo Integrado de Media Móvil) y SARIMA (ARIMA Estacional).

Tabla Resumen de Modelos Clásicos:

Modelo	Descripción	Componentes Clave	Uso Típico
AR (Autorregresivo)	Modelo que utiliza la dependencia entre una observación y un número de observaciones pasadas.	Orden del modelo (p)	Series con influencia fuerte y directa de valores pasados.
MA (Media Móvil)	Modelo que utiliza la dependencia entre una observación y un ruido residual de observaciones pasadas.	Orden del modelo (q)	Series donde el ruido o los choques aleatorios son factores importantes.
ARMA	Combinación de AR y MA. Utiliza tanto las observaciones pasadas como el ruido residual.	Orden de AR (p), Orden de MA (q)	Series donde ambos, valores pasados y ruido, son relevantes.
ARIMA	Extensión de ARMA que incluye la integración para hacer la serie temporal estacionaria.	Orden de AR (p), Orden de Diferenciación (d), Orden de MA (q)	Series no estacionarias que requieren diferenciación.
SARIMA	Versión estacional de ARIMA. Incluye componentes estacionales además de los no estacionales.	Órdenes de AR, MA y Diferenciación tanto para componentes estacionales como no estacionales.	Series con patrones estacionales claros.

Modelos AR(Autorregresivos):

Definición:

El método AR es uno de los modelos más sencillos dentro de los modelos clásicos. Se utiliza en el análisis de series temporales para predecir valores futuros en función de los valores previos de la serie. El modelo AR utiliza regresión lineal para modelar la relación entre el valor actual de la serie y los valores anteriores, y se basa en la suposición de que los valores anteriores de la serie son buenos predictores de los valores futuros.

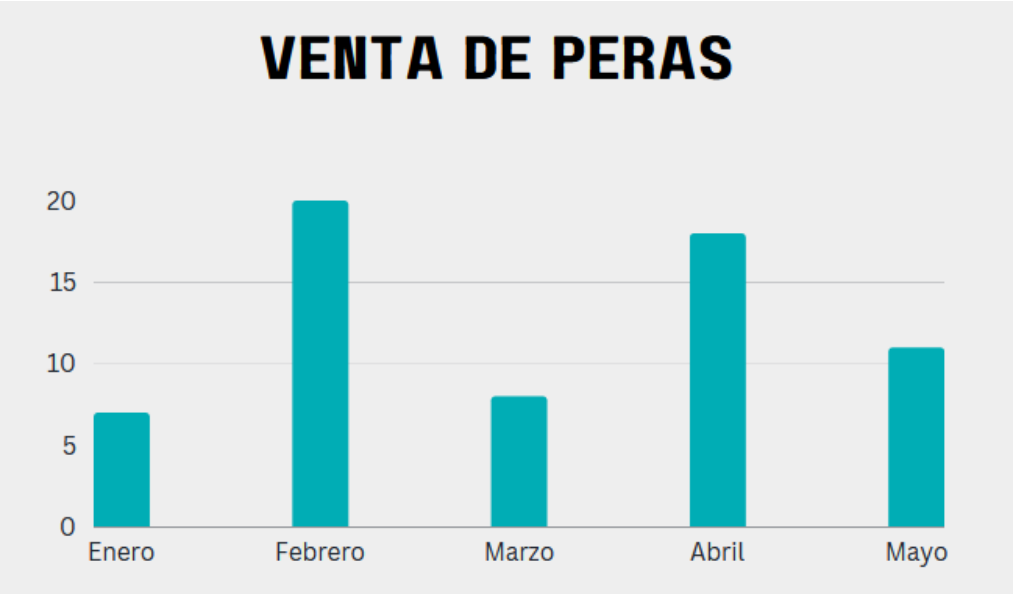
Para poder situarnos y continuar con este modelo, vamos a explicar dos términos que van a ser claves para entender este primer apartado:

-Autorregresión.

-ACF/PACF

En primer lugar, vamos con la función que da nombre a nuestro modelo, la autoregresión. Lo primero en lo que tenemos que hacer hincapié es en la primera parte de nuestra palabra,"auto", esto nos indica que no es una regresión al uso, en la cual, intentamos predecir un valor "x" en base a otros valores "y,w,z…" ; en este caso la nomenclatura "auto" nos viene a indicar que la predicción del valor "x" vendrá dada por los valores anteriores de la serie, es decir, que se prevé que los valores anteriores de nuestro objetivo tengan influencia directa en el valor actual a calcular.

Pongamos un ejemplo gráfico para ilustrar la necesidad que estamos buscando representar con este modelo, por ejemplo, supongamos una gráfica que indica la cantidad de peras vendidas en un pueblo. Tenemos la siguiente gráfica:

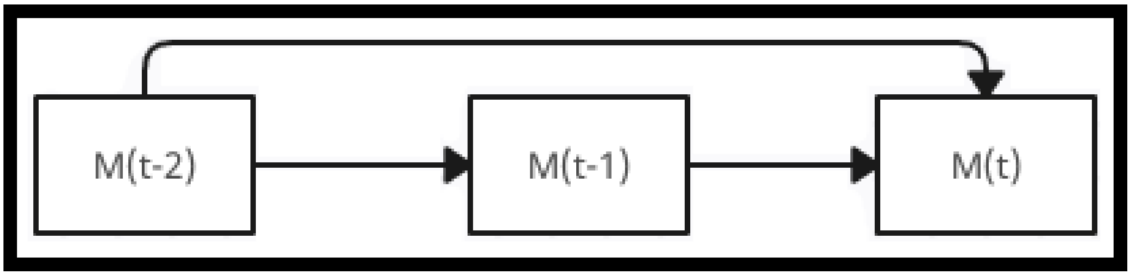


Vemos como tenemos la gráfica de la venta de peras y en cada uno de los primeros cinco meses del año, vemos que enero se vendieron 7 peras, en febrero 20 y así sucesivamente. Como hemos venido exponiendo, buscamos obtener el valor de peras en el siguiente instante de tiempo Mes(t), que sería junio en nuestro caso, basándonos en los valores anteriores, ya que es un modelo autorregresivo. Con lo cual, Mes(t) vendrá influenciado en mayor o menor medida por Mes(t-1), Mes(t-2), …. Mes(t-n), siendo n un instante de tiempo anterior. Otra cuestión que nos incumbe ahora es, ¿Cómo se si el valor o valores anteriores afectan directa o indirectamente a nuestro M(t)? Bien, en este caso tenemos que hacer un paréntesis para hablar del segundo punto mencionado en el inicio de este apartado, la ACF/PACF.

Función de Autocorrelación (ACF) y Función de Autocorrelación Parcial (PACF):

La idea con este apartado es determinar qué valores anteriores influyen en el valor actual, y cuáles no. Por ejemplo, en la gráfica anterior vemos como en los meses impares, enero, marzo y mayo, los valores no parecen tener que ver con los vendidos en febrero y abril. Esto puede ser debido a un suceso en el pueblo, como que, en los meses pares del año, sea típico realizar una receta típica que incluya a la mencionada fruta.

Este pequeño comentario solo nos quiere indicar que podemos tener valores anteriores en el tiempo M(t-n) que entorpezcan al modelo a la hora de calcular el instante actual M(t). Para esto tenemos en primer lugar la Función de Autocorrelación:



Siendo M(t) -> Junio, M(t-1) -> Mayo, M(t-2) -> Abril. El fin de esta gráfica es representar las dos maneras que tenemos de alcanzar M(t), o visto de otra manera, las dos maneras por las que se ve influenciada nuestra variable en el instante de tiempo (t).

En la representación gráfica, vemos que tenemos dos posibles caminos por los que la variable M(t-2) puede afectar a M(t), uno directo y otro indirecto. M(t-2) puede afectar directamente al instante actual M(t), y también puede afectar al mes anterior, M(t-1) y de esta manera, indirectamente, afectar al instante actual M(t). Estas dos relaciones juntas son las que dan forma al Función de Autocorrelación (ACF) para la correlación de la venta de peras en abril y la venta de peras en junio.

Ahora vamos a tratar la Función de Autocorrelación Parcial (PACF). El punto más importante a tener en cuenta en relación con el ACF es que aquí solo tenemos en cuenta el efecto directo, no nos importa el efecto que tenga M(t-2) en otros instantes de tiempo relativos al instante actual. En definitiva, solo nos importa:

M(t-2) -> M(t)

¿Por qué nos interesa obviar el componente indirecto? Por la naturaleza de los datos. Intencionalmente, antes mencionamos que en la gráfica había una fuerte relación entre los meses pares e impares, ya que en los impares se realizaba una receta típica en el pueblo que impulsaba la venta de peras; esto implica que la venta de fruta en junio no vendrá influenciada por la venta de fruta en mayo, o no en gran medida. La fórmula matemática que representa este paradigma es la siguiente:

$$M_t = \beta_0 + \beta_1 * M_{t-1} * + \beta_2 * M_{t-2} * + \beta_3 * M_{t-3} * + \beta_n * M_{t-n}$$

Esta fórmula consigue aislar el componente indirecto del directo. Pongamos como supuesto que queremos calcular nuestro PACF para el valor k = 2, siendo k nuestro instante anterior en el tiempo. En nuestra fórmula anterior, esto vendría representado por el siguiente término, donde el PACF viene dado por el término β_2.

$$+ \beta_2 * M_{t-2}$$

Podemos decir que esta fórmula nos permite obviar la componente directa porque el efecto del instante anterior en nuestra serie temporal, k=1, viene representado por el término previo de la misma.

En general, los modelos AR pueden ser bastante efectivos para predecir series temporales estacionarias (es decir, aquellas cuyas características estadísticas no cambian con el tiempo). Sin embargo, pueden no ser tan efectivos para predecir series temporales no estacionarias (aquellas cuyas características estadísticas cambian con el tiempo), ya que no tienen en cuenta la tendencia o la estacionalidad.

MA

Definición:

(hay correcciones en la parte de arriba, pero no había activado el control de cambios

El modelo MA (Moving Average) es un modelo popular en el análisis de series temporales que se utiliza para modelar la media móvil de un proceso estocástico ,comúnmente en el análisis de series temporales para predecir futuros valores de la serie.

En primer lugar abordaremos el objetivo principal de la MA. La media móvil se basa en el error pasado para predecir el futuro, la pregunta sería, en cuánto nos hemos equivocado en el instante anterior, para, de esa forma, determinar un valor que se aleje menos en la siguiente iteración de nuestra serie temporal. Para conseguir entender cómo funciona este modelo, vamos a recurrir a un ejemplo similar al utilizado con el modelo autorregresivo.

Volvemos a situarnos en el mismo pueblo, donde tienen una industrial local de postres a base de peras , en este caso calcularemos, en el periodo de una semana, la cantidad de peras que se necesitan. Para ello utilizaremos la siguiente fórmula:

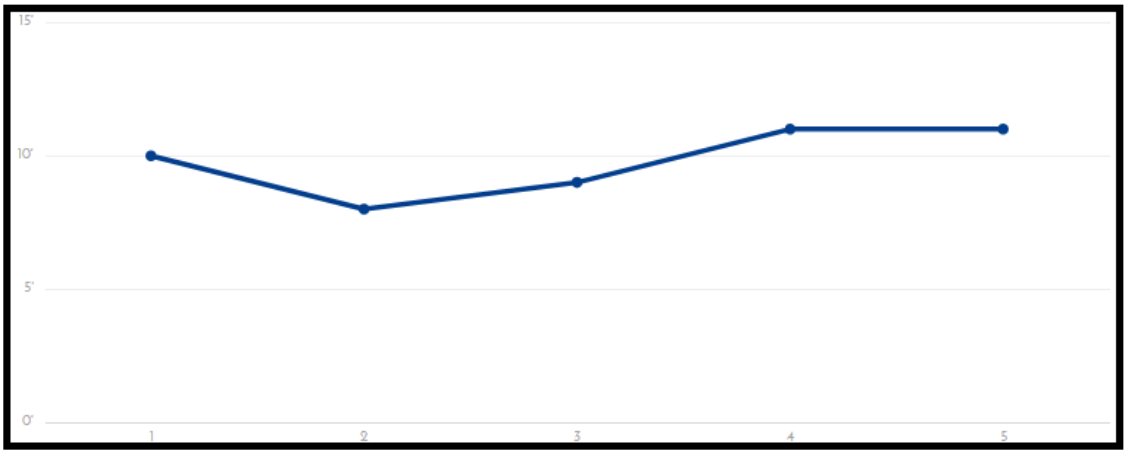
$$M_t = \beta + \phi * \varepsilon_{t-1}$$

M_t, vuelve a ser nuestro valor a predecir. En esta ocasión tenemos dos términos nuevos β y ε. β representa la media esperada , por ejemplo, supongamos que al inicio de la semana es de 10 unidades, con lo que nuestro término β sería igual a 10. Por otro lado, el término ε_{t-1} indica el "error" en el instante t-1. Por ejemplo, si la media es de 10 unidades, y nos sobran 3, nuestro error sería de -3, ya que nos han sobrado 3 para llegar a la cantidad objetivo que sería 7. Por último, el término *Phi* que acompaña a épsilon representa la importancia que le damos al error anterior en el cálculo de

nuestro valor actual. Vamos a ilustrarlo con un ejemplo, para el siguiente ejemplo, el valor de Phi utilizado es 0,5:

t	M(t)	ε	M(t) REAL
1	10	-3	7
2	8´5	1	9`5
3	9	0	9
4	9	2	11
5	10	1	11

Esta tabla representa el comportamiento de la media móvil. En el primer instante partimos de una media de 10 unidades, como podemos ver en la primera casilla, al final del día vemos que nos han sobrado 3 unidades por vender, con lo cual tenemos un error épsilon de 3, esto nos indica que la cantidad idónea hubieran sido 7 unidades. Vamos ahora al segundo instante de tiempo, donde tras realizar el cálculo del error "-3" multiplicado por el valor de Phi -> 0,5 obtenemos que M(t) será igual a 10 – 1.5 = 8,5. Ahora al final del día vemos que nos ha faltado 1 unidad para poder satisfacer las necesidades del pueblo, y por eso el error es positivo y obtenemos que hubiéramos necesitado 9'5 unidades para satisfacer las necesidades de los habitantes. Si seguimos realizando este planteamiento conseguiremos rellenar la tabla tal y como está en el ejemplo. Antes de pasar al siguiente apartado, vamos a ilustrar con una gráfica estos valores para enfatizar una cualidad de este modelo.



Vemos como hay una tendencia en base a la media, que en este caso es igual a 10, como su nombre indica la media móvil siempre va a fluctuar sobre la media. Finalmente, la fórmula y los cálculos realizados anteriormente eran para el modelo más sencillo y fácil de representar gráficamente MA(1), la fórmula generalizada para el modelo MA(n) sería la siguiente:

$$M_t = \beta + \phi_1 * \varepsilon_{t-1} + \phi_2 * \varepsilon_{t-2} + \phi_3 * \varepsilon_{t-3} + \phi_n * \varepsilon_{t-n}$$

Similar a lo que ocurría con el PACF, simplemente tenemos en cuenta más instantes anteriores para calcular M(t), cada instante tiene una Phi independiente que le dará mayor o menos “peso” al término de error que acompañe.

Ahora, volvemos a la parte relativa al modelo, En MA la predicción del siguiente valor en la serie depende de una combinación lineal de los errores de predicción anteriores. La idea detrás del modelo MA es que los errores de predicción pasados son una buena indicación de los errores futuros, por lo que al modelar la serie de tiempo en términos de estos errores se puede obtener una buena predicción de los valores futuros.

El modelo MA es un modelo lineal y se puede expresar como una suma ponderada de los errores de predicción pasados. El "orden" del modelo MA se refiere al número de términos de error que se incluyen en el modelo. Un modelo MA de orden 1 utiliza un solo término de error para predecir el siguiente valor en la serie, un modelo MA de orden 2, los dos anteriores, etc.

ARMA

Definición:

El modelo autorregresivo de media móvil (ARMA) es una extensión del modelo de media móvil (MA) y del modelo autorregresivo (AR)que combina ambos modelos en uno solo. Con esta combinación podemos modelar la dependencia autorregresiva y la dependencia de media móvil simultáneamente.

Podemos apreciar esta combinación de modelos en la correspondiente:

$$M_t = \beta_0 + \beta_1 * M_{t-1} + \phi_1 * \varepsilon_{t-1}$$

Como podemos deducir, la ecuación ecuación es la compuesta por la parte AR y MA vistas anteriormente, en este caso representa un modelo ARMA(1,1). La primera parte representa la parte autorregresiva, mientras que la segunda es la vista en el apartado anterior, la media móvil. La ecuación de un ARMA(2,1), por ejemplo, no sería más que añadir un término adicional en la parte autorregresiva, mientras que para un modelo ARMA(1,2) bastaría con añadir otro término de media móvil.

Para hilar más fino y determinar que componentes de nuestra serie temporal influyen o no en este modelo, hay que recordar las componentes PACF y ACF.

Como ya vimos, la función de autocorrelación ACF es la que determina el componente directo e indirecto de los instantes anteriores con el instante actual de nuestra serie (parte MA delmodelo), mientras que en la parte AR utilizamos la función de autocorrelación PACF que solo tiene en cuenta el componente directo, es decir, no tiene en cuenta la influencia de M(t-2) en M(t-1).

En definitiva, ayudándonos de las funciones ACF y PACF podríamos obtener mucho mejor nuestro resultado de modelo ARMA, ya que podemos ajustar por separado la componente móvil de la autorregresiva, dependiendo de la naturaleza del problema.

ARIMA

Definición:

El modelo ARIMA, por su parte, es una extensión del modelo ARMA que agrega la capacidad de modelar tendencias no estacionarias en la serie temporal.

La "I" en el modelo ARIMA se refiere a la parte de "Integración" del modelo. En un modelo ARIMA (p, d, q), la "d" representa el número de veces que debemos diferenciar la serie temporal para hacerla estacionaria, lo que se conoce como el "orden de integración". Por lo tanto, la "I" en ARIMA se refiere a la cantidad de veces que se ha aplicado la diferencia a la serie temporal para que sea estacionaria antes de aplicar el modelo ARMA, visto anteriormente.

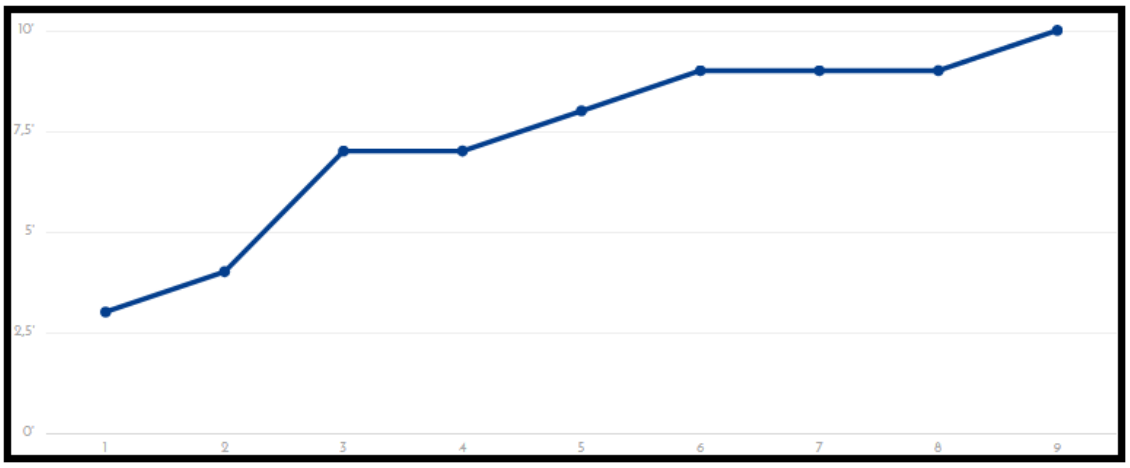
El modelo ARIMA se tiende expresar matemáticamente como una combinación de los modelos vistos en los apartados anteriores:

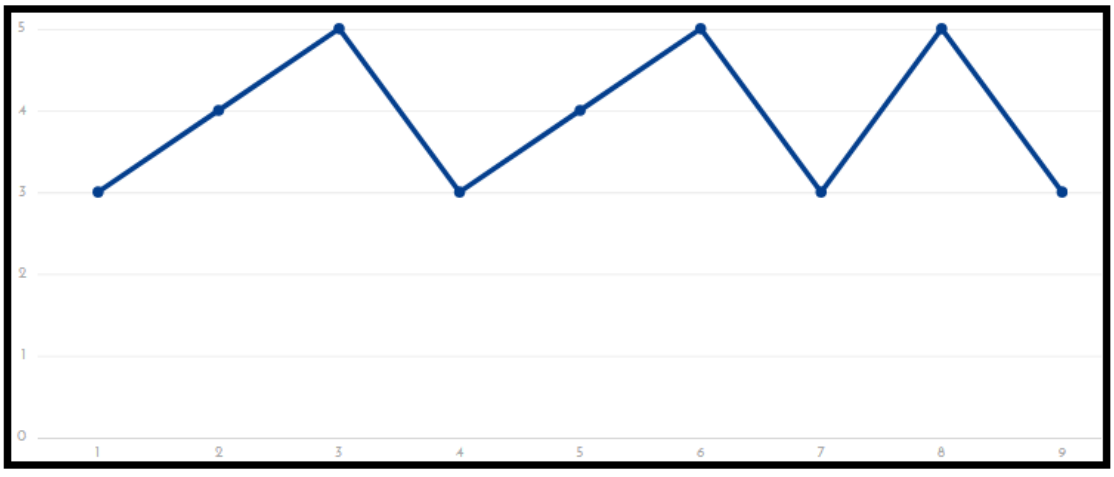
-Un modelo autoregresivo (AR),

-Un modelo integrado (I)

-Un modelo de media móvil (MA).

¿Qué significa esto? Veamos un ejemplo gráfico que ilustre este paradigma.





Los valores representados son completamente aleatorios, y solo sirven para visualizar el problema que viene a solucionar. Como podemos observar en la primera gráfica, no tenemos estacionalidad y vemos como tiene una tendencia ascendente, esto para los modelos que hemos visto es algo negativo, ya que los valores anteriores no serán tan buenos para la predicción. Sin embargo, en la segunda gráfica vemos que la gráfica fluctúa, pero no tiene una componente que la haga "no estacionaria".

Vamos a ilustrar las gráficas expuestas con un ejemplo, la venta de peras nuevamente. En esta ocasión, al no ser estacionaria, vamos a suponer que la población en el pueblo aumenta en 2 personas cada mes, con lo que la demanda de unidades de peras tendrá un crecimiento similar, esto la convierte en no estacionaria, nuestros modelos AR, MA y ARMA ya no son idóneos.

Por esta razón nace la "I" del modelo ARIMA, que realiza la integración necesaria para pasar de una gráfica como la primera a otra como la segunda, obviamente, esta sigue teniendo sus fluctuaciones, pero ya es algo para lo que el modelo está preparado, un modelo estacionario con una media constante.

La integración intenta predecir la serie completa, la cual tiene una componente que hace que no sea estacionaria, vamos a centrarnos en una "fracción" de la misma. Por ejemplo, las ventas en un mes "t" y en el mes "t-1", con lo cual tendríamos lo siguiente:

$$Z_t = a_{t+1} - a_t$$

Ahora, una vez visto esto, vamos a definir el modelo más básico de ARIMA, el ARIMA (1,1,1), que tendría la siguiente forma:

$$Z_t = \beta_1 * Z_{t-1} + \phi_1 * \varepsilon_{t-1} + \varepsilon_t$$

Podemos observar que estamos intentando predecir la diferencia del número de peras vendidas entre "t+1" y "t". Y para ello aplicamos un ARMA, que es lo que vemos en la segunda fórmula planteada. No obstante, ahora tenemos que realizar el paso inverso, ya que nosotros lo que queremos es obtener el número de unidades de peras, no Z(t).

Para volver al plano no estacionario, tomando como referencia la nomenclatura utilizada en la primera fórmula. Vamos a suponer que nuestra serie temporal va desde a(0) hasta a(L). Con este supuesto, vamos a transformar la ecuación de la que queremos despejar a(k), siendo k = t+1, con lo que se quedaría de la siguiente forma:

$$a_k = Z_{k-1} - a_{k-1}$$

Al tener k = t+1, t sería igual a k-1, con lo que despejamos y dejamos a(k) a la izquierda de la ecuación. Una vez hecho esto, repetimos el proceso para el instante anterior, con lo que tendríamos:

$$a_k = Z_{k-1} + Z_{k-2} - a_{k-2}$$

Hemos sustituido a(k-1) siguiendo el procedimiento explicado y obtenemos esa ecuación, si continuamos iterando obtenermos:

$$a_k = \sum_{i=1}^{k=l} Z_{k-i} + a_l$$

Con esto ya tenemos una manera de obtener el valor a predecir, a(k). Y disponemos de todos los datos necesarios para realizar esta predicción, siendo el sumatorio de Z(k-i), cada instante anterior pronosticado por la fórmula del ARMA sobre Z(t), la segunda fórmula comentada en este apartado. Por último, siendo a(L) el último valor conocido de nuestra serie temporal.

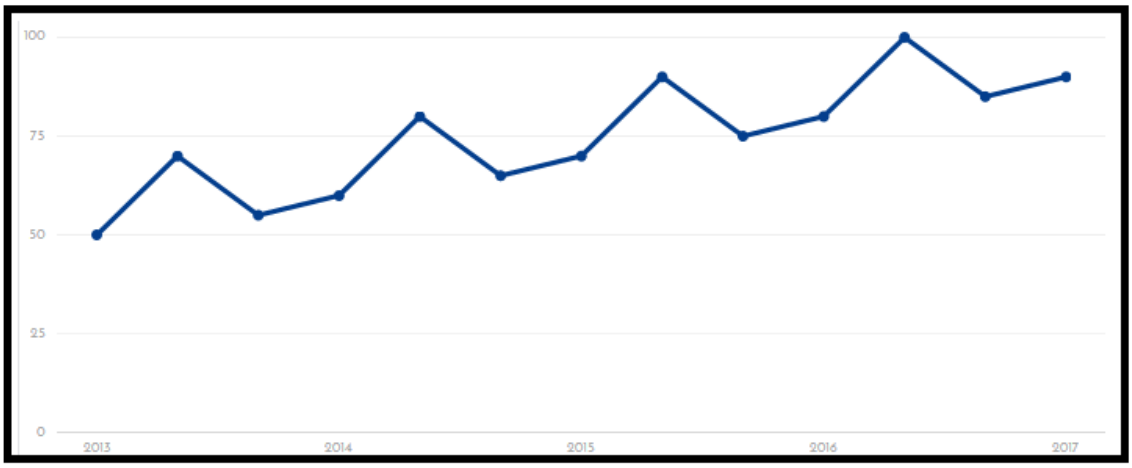
SARIMA

Definición:

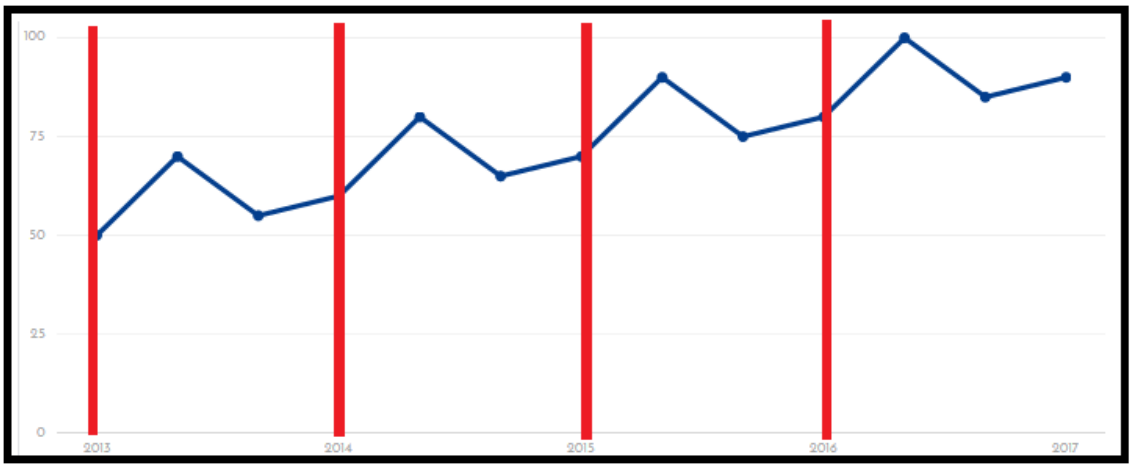
El modelo autorregresivo integrado de media móvil estacional (SARIMA) es una extensión del modelo ARIMA que agrega la capacidad de modelar la estacionalidad en la serie de tiempo. El modelo SARIMA se utiliza para modelar una serie de tiempo que tiene una estacionalidad clara y una tendencia no estacionaria.

Al igual que el modelo ARIMA, el modelo SARIMA combina un modelo autorregresivo (AR), un modelo integrado (I) y un modelo de media móvil (MA). Además, el modelo SARIMA incluye un componente estacional que se utiliza para modelar la estacionalidad en la serie de tiempo.

Para ilustrar este último modelo clásico, vamos a obviar los ejemplos como en modelos anteriores ya que serían muy similares y vamos a centrarnos en por qué utilizamos un modelo SARIMA en una serie temporal.



En la anterior gráfica, vemos lo siguiente, en primer lugar, que tiene una tendencia lineal, cuya media no es constante, con lo cual sugiere un modelo ARIMA, pero a su vez si nos fijamos detenidamente, vemos que, a lo largo del año, hay una repetición o semejanza en los valores.



Y es que esa figura parecida a una “V” invertida, se repite a lo largo de todos los años, teniendo en cuenta que es un modelo no estacionario. Con lo cual, cuando nos referimos a estacionalidad de una serie temporal, nos referimos a esta repetición que aparece dividida en las franjas rojas, en este caso podría tratarse de una estacionalidad anual.

Como hemos venido haciendo con todos los modelos, vamos a revisar la fórmula matemática que da vida a este modelo, en este caso, vamos a estudiar la más básica, el SARIMA$(1,1,1)(1,1,1)_3$. Antes de entrar con la parte matemática, vamos a explicar brevemente los parámetros que influyen en nuestro modelo, la mayoría son bien conocidos, SARIMA(p,d,q)(P,D,Q)m. Los parámetros “p”, “d” y “q” son los mismos que para el modelo ARIMA, en cuanto a los siguientes parámetros, son básicamente análogos, pero en el contexto estacionario, por eso vamos a hablar primero del parámetro“m”. La “m” indica la estacionalidad del modelo, el número de veces que se repite, en este caso sería tres, debido a que en un año hay un periodo de tres valores como se puede apreciar en la gráfica anterior. Finalmente, los valores “P”, “D” y “Q”

son los mismos que los “p”, “d”, “q” pero en el contexto anual, en otras palabras, en el contexto estacionario del modelo.

Para ilustrar esto disponemos de lo siguiente, para un modelo SARIMA(1,1,1)(1,1,1)$_3$:

$$(1-\phi_1\beta)\left(1-\phi_1\beta^3\right)(1-\beta)\left(1-\beta^3\right)y_t = (1+\phi_1{*}\beta)\left(1+\phi_1{*}\beta^3\right)\varepsilon_t$$

Si vamos de izquierda a derecha tenemos lo siguiente, el primer paréntesis de la ecuación refleja el parámetro “p”, que quiere conseguir tener en cuenta la serie temporal hace un “periodo” para realizar la nueva predicción. El siguiente término, como vemos es casi idéntico al anterior, lo que nos indica que es la “P” de la parte estacionaria y está elevado a tres debido al valor “m” del que ya hablamos anteriormente. Si seguimos, los dos siguientes términos son los relativos a la parte integrada, es decir, “d” y “D”. Y finalmente, al otro lado del igual, disponemos de lo relativo a los parámetros “q” y “Q”.

Vamos a expandir esta ecuación para trabajar correctamente, no obstante, vamos a utilizar un modelo más reducido para comprender mejor el procedimiento a seguir antes de adentrarnos con el SARIMA(1,1,1)(1,1,1)$_3$. En este caso, vamos a utilizar SARIMA(1,0,0)(0,1,1)$_3$:

$$\begin{gathered}
(1-\phi_1\beta)\left(1-\beta^3\right)y_t = \left(1+w_1{*}\beta^3\right)\varepsilon_t \Longrightarrow \\
\left(1-\phi_1\beta-\beta^3+\phi_1\beta^4\right)y_t = \varepsilon_t + w_1{*}\varepsilon_{t-3} \Longrightarrow \\
y_t - y_{t-3} = \phi_1 y_{t-1} - \phi_1{*}\varepsilon_{t-4} + w_1{*}\varepsilon_{t-3} + \varepsilon_t \\
Z_t = y_t - y_{t-3} \Longrightarrow \\
Z_t = \phi_1 Z_{t-1} + w_1{*}\varepsilon_{t-3} + \varepsilon_t
\end{gathered}$$

Los modelos SARIMA son útiles para modelar datos de series temporales que tienen una estacionalidad clara, como ventas minoristas mensuales o trimestrales, datos climáticos estacionales, entre otros. Los modelos SARIMA pueden proporcionar una predicción más precisa de los valores futuros de la serie de tiempo que los modelos ARIMA estándar, especialmente cuando hay una estacionalidad clara en los datos.

Primeras comparativas de los modelos clásicos:

A continuación aplicaremos estos modelos a un conjunto de datos, en primer lugar, los usaremos con los datos de consumo energético que se pueden dar en una vivienda familiar, en un periodo de tiempo determinado. Este “dataset” tiene el nombre de “Household_power_consumption” y mide el consumo en horas del día, no obstante,

para este primer acercamiento, vamos a trabajar con una variante que divide los datos en el eje temporal en días.

El conjunto de datos" cuenta con 9 parámetros, representados por 9 columnas: :

-Datetime: parámetro que indica la fecha en formato YYYY-MM-DD de cuando fue tomada la muestra.

-Global_active_power: parámetro que nos da la información relativa al consumo activo de la energía en ese momento de tiempo en el hogar.

-Global_reactive_power: columna encargada de referenciar los datos que hacen referencia al consumo reactivo de energía que tiene la vivienda en el instante de tiempo.

-Global_intensity: parámetro relativo a la intensidad de la corriente en ese instante de tiempo.

-Sub_metering_1,2,3,4:

El objetivo principal consiste en analizar patrones de consumo e intentar realizar una predicción del comportamiento en base a tendencias e influencias en el consumo eléctrico del hogar.

Muestra del conjunto de datos:

A continuación, procedemos a mostrar el al formato de los datos descritos anteriormente para situar las magnitudes y valores aproximados con los que vamos a tratar.

	datetime	Global_active_power	Global_reactive_power	Voltage	Global_intensity
0	2006-12-16	1209.176	34.922	93552.53	5180.8
1	2006-12-17	3390.460	226.006	345725.32	14398.6
2	2006-12-18	2203.826	161.792	347373.64	9247.2
3	2006-12-19	1666.194	150.942	348479.01	7094.0
4	2006-12-20	2225.748	160.998	348923.61	9313.0

Como se puede observar cada fila representa un día que viene dado por la columna "datetime".

Por otro lado, tenemos los "sub_metering", que, por formato de este documento, han sido adjuntados en una imagen a parte:

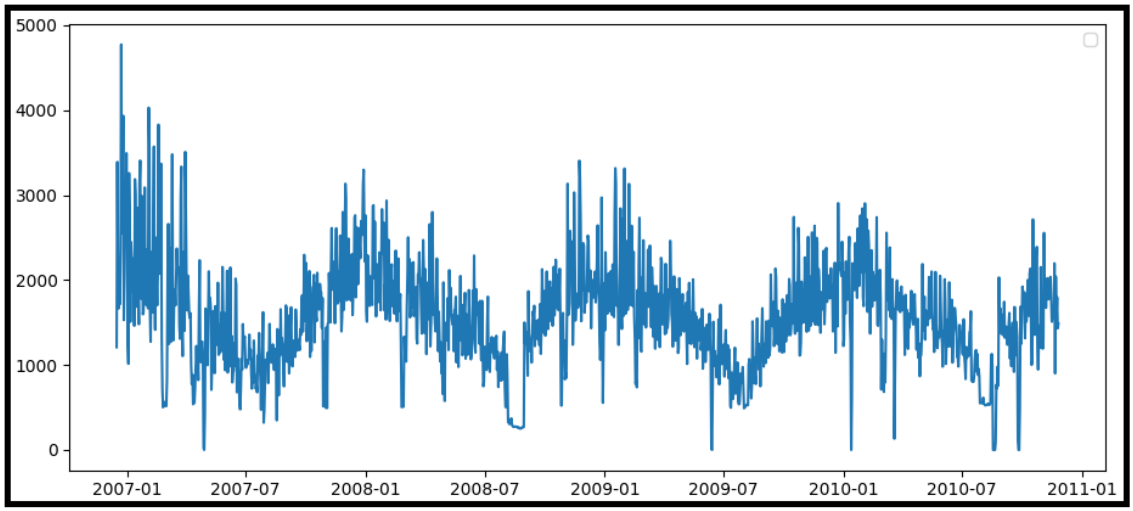
Sub_metering_1	Sub_metering_2	Sub_metering_3	Sub_metering_4
0.0	546.0	4926.0	14680.933319
2033.0	4187.0	13341.0	36946.666732
1063.0	2621.0	14018.0	19028.433281
839.0	7602.0	6197.0	13131.900043
0.0	2648.0	14063.0	20384.800011

Análisis del dataset:

Como hemos descrito en el estudio previo de los modelos clásicos, estos modelos son dependientes de la naturaleza de los datos. Los modelos AR, MA y ARMA necesitan series estacionarias para que funcionen correctamente, ARIMA depende de una subida constante en la función para su correcta integración.

En este apartado, vamos a observar cómo se comportan los datos previamente al entrenamiento de nuestros modelos, y así conseguir sacar un mayor partido de estas. Con esto buscamos no ir a ciegas a la hora de aplicar o no un modelo, de sacar o no unas conclusiones basadas en los meros resultados obtenidos.

En primer lugar, vamos a representar nuestra variable objetivo “Global_active_power” a lo largo del tiempo:



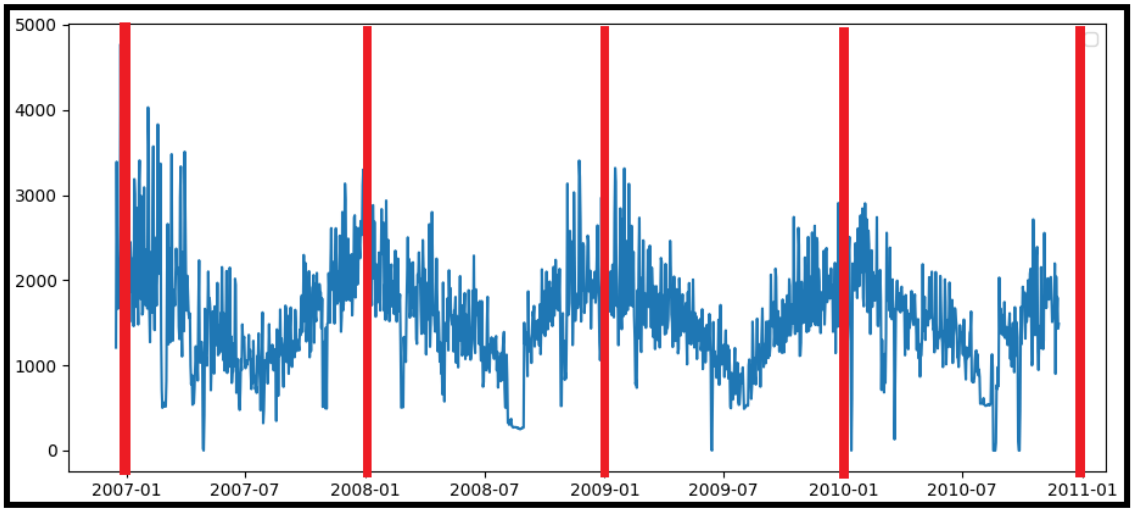
En base a lo comentado anteriormente, vamos a ir punto por punto analizando la información que nos arroja esta representación gráfica de nuestra variable objetivo.

En primer lugar, ¿es estacionaria? Sí, como podemos ver la serie tiende a tener una media constante que no varía a lo largo del tiempo. Recordemos que una serie no estacionaria es aquella que tiene un crecimiento positivo o negativo a lo largo del tiempo, esto influye en nuestros modelos clásicos debido al tratamiento de los datos anteriores. A modo de repaso, nuestro modelo AR utiliza los datos anteriores para predecir los siguientes, si la serie no fuera estacionaria en el tiempo, este modelo no la tendría en cuenta y dispondríamos de un error mayor en nuestra predicción. En este caso, podemos aplicar AR, MA y ARMA sin problema ya que no tenemos ese contratiempo.

¿Tiene sentido que sea estacionario el consumo en un hogar? Sí, en periodos cortos como son 4 años en una viviendo, el consumo energético no suele variar en exceso. Claro está que, si se realizara un estudio que abarcara los últimos 30 años, si, se podría apreciar un crecimiento y podríamos hablar de una serie no estacionaria.

Otro punto interesante para observar sería la estacionalidad de la serie, en este caso nos referimos a la repetición en un lapso temporal de los valores de la serie. Podemos

observar que si existe una repetición de carácter anual a modo de “v”. La podemos apreciar mejor en la siguiente figura:



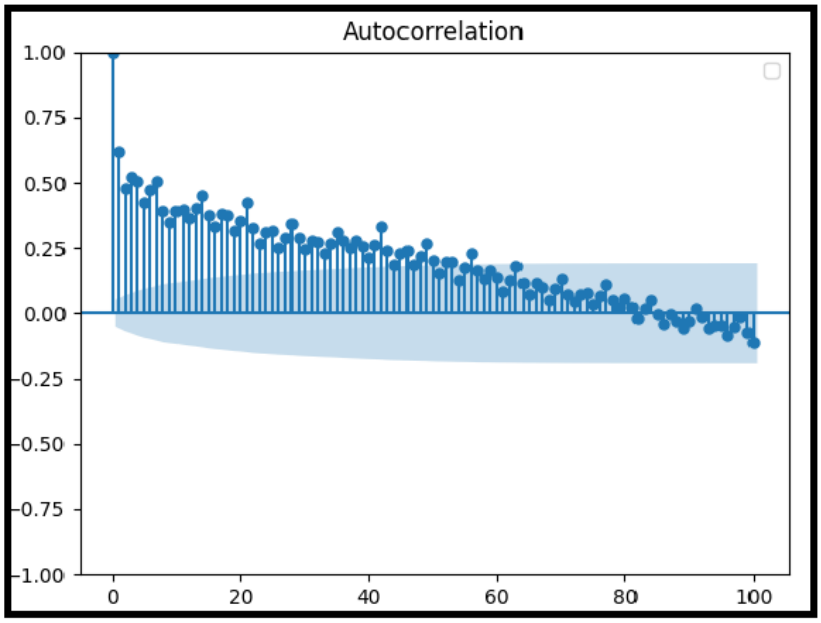
La estacionalidad, no es más que la repetición de un patrón a lo largo de nuestra serie temporal. Para aprovechar esta característica del conjunto de datos, disponemos del modelo SARIMA, que se encarga de tener en cuenta este factor a la hora de realizar la predicción.

¿Es coherente una estacionalidad anual en la serie? Sí, el consumo eléctrico en un hogar varía dependiendo de la estación y clima a la que está sujeta. En invierno es cuando vemos un mayor consumo debido a que las temperaturas por lo general son inferiores y es necesario utilizar calefacción, días más cortos, con lo cual se utiliza en mayor medida el alumbrado, y un largo etc.

Finalmente, vamos a realizar un estudio de las componentes ACF y PACF. Este punto es crucial para determinar el punto de salida y posible comportamiento de nuestro modelo en base al elegido, la componente ACF va ligada a la parte autorregresiva, es decir, si la función de autocorrelación arroja buenos datos el modelo AR será idóneo. Mientras que la PACF va ligada al modelo de la media móvil o MA.

Para codificar el gráfico del ACF, utilizamos el siguiente código:

```
acf_plot = plot_acf(dataset["Global_active_power"], lags = 100)
```

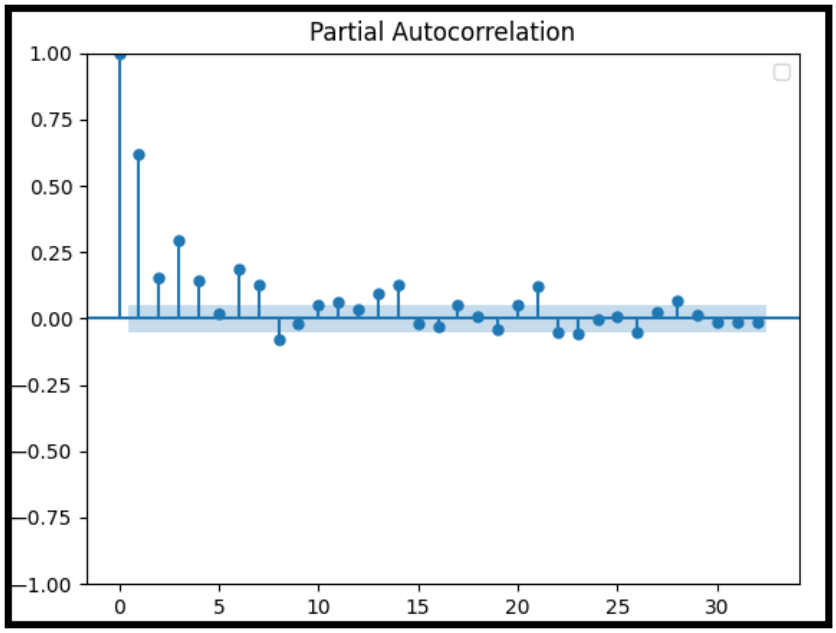


Gráfica de la Autocorrelación

En primer lugar, obviemos el primer valor, ya que es el valor inicial y va a ser 1 siempre. Continuando con el análisis, ¿Qué debemos tener en cuenta sobre esta gráfica? Que tiende a cero y que todo valor dentro del espacio azul se puede considerar 0. En base a los dos puntos comentados, podemos intuir que estamos tratando con un proceso Autorregresivo.

No obstante, antes de afirmar esto rotundamente, vemos a realizar esta misma representación con el PACF:

```
pacf_plot=plot_pacf(dataset["Global_active_power"])
```



gráfica de la Autocorrelación Parcial

En esta ocasión vuelve a suceder lo mismo, las componentes que se sitúen en el interior de la franja de color azul, son indiferentes para nuestro modelo y pueden considerarse 0. En este caso, lo que nos indica esta representación son los “pasos atrás” con los que podría ser bueno que empezara el modelo AR, en este caso, podría ser el (1,2,3,4,6,7,13,14). Como hemos visto, el modelo AR que mejores datos nos arroja es cuando utilizamos un “lag” igual a 7.

El breve estudio que hemos realizado ahora solo es la punta del iceberg, y tenemos que ver esta información como un punto de partida para empezar a trabajar, no como un factor que excluya o de una falsa información sobre el conjunto de datos. A continuación mostramos la representación visual del resto de parámetros:

Global_reactive_power:

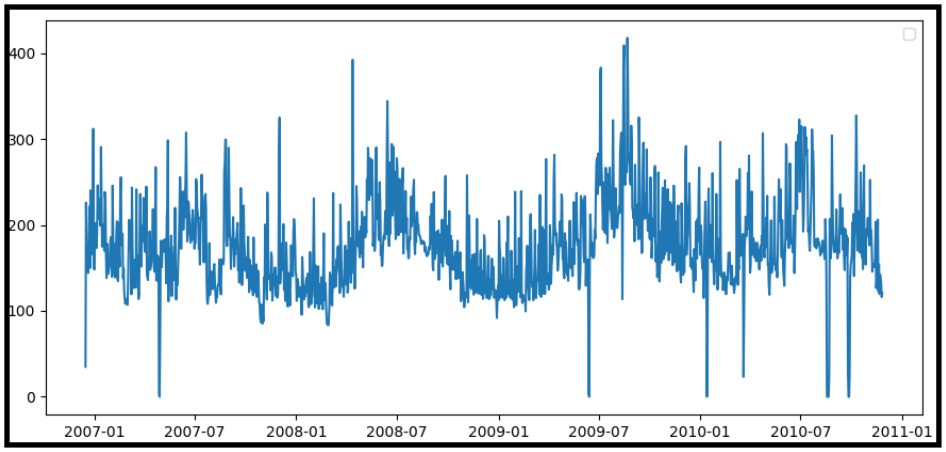


Ilustración variable objetivo

Voltage:

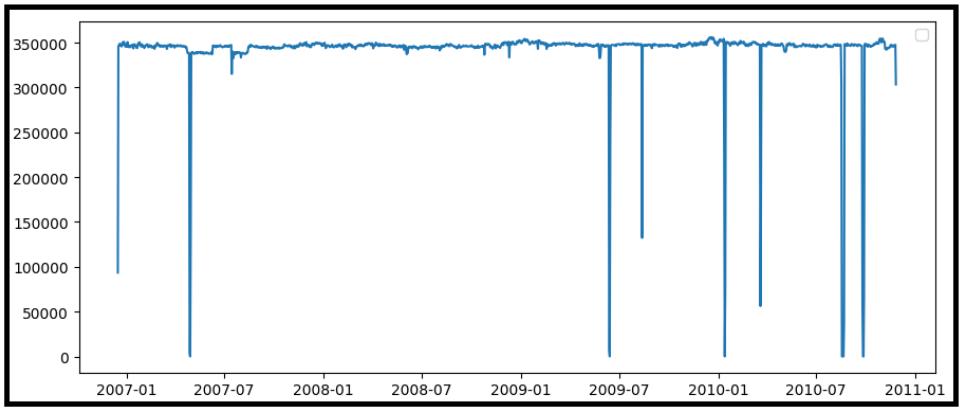


Ilustración variable “Voltaje”

Global_intensity:

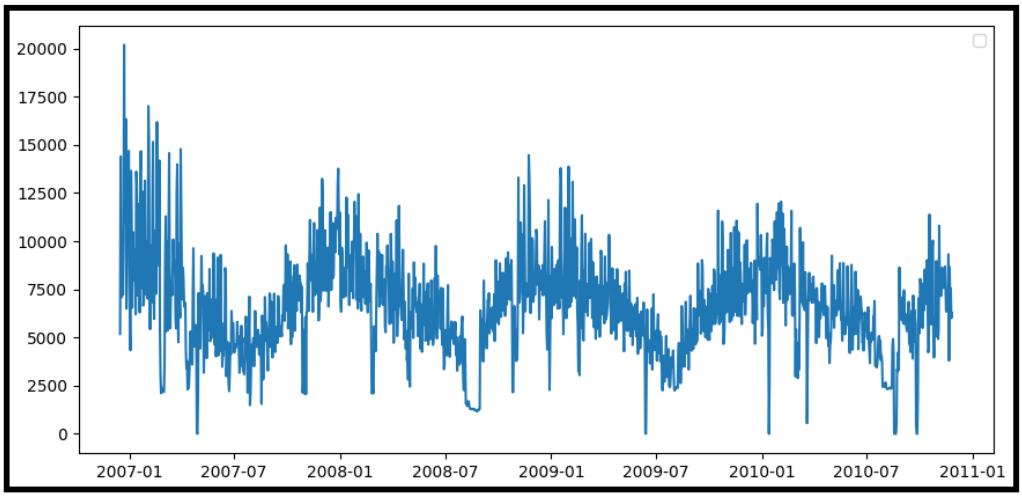


Ilustración variable "Intensidad"

Sub_metering:

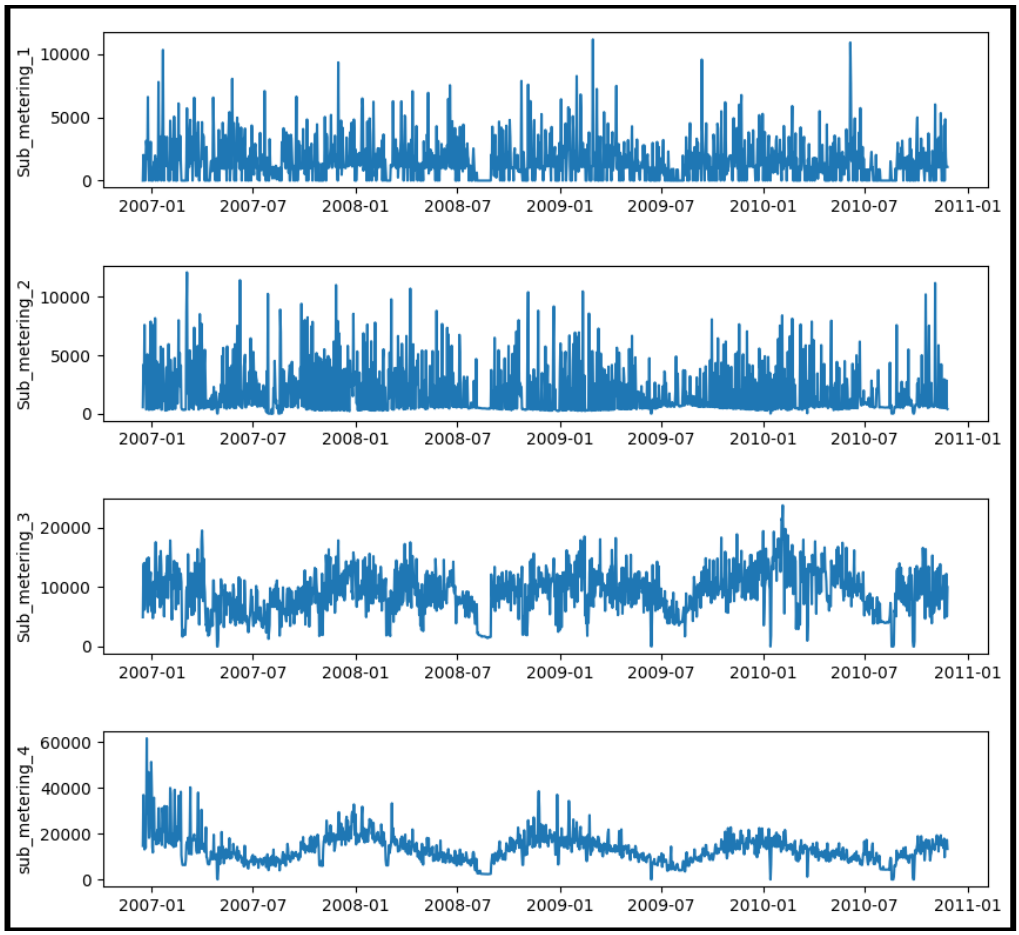


Ilustración variables "sub_metering"

Implementación en Python:

En el siguiente punto de nuestra memoria, vamos a comentar lo relativo a la implementación en Python de los modelos clásicos ya mencionados previamente en este documento. Para realizar esta explicación vamos a ir comentando a grandes rasgos las funcionalidades de cada método y el propósito que tienen para realizar nuestro propósito.

Split_dataset:

La siguiente función es la encargada de realizar la separación entre los conjuntos de entrenamiento y validación de nuestros datos. Esto es necesario para posteriormente poder entrenar la red neuronal y validar nuestros resultados con unos valores nunca vistos por nuestro modelo.

Por último, se realiza una subdivisión del conjunto de datos, tanto de “train” como “test” en subconjuntos de siete elementos. Esto se realiza debido a que la predicción que queremos aportar es de una semana, que es un conjunto temporal de siete días.

```
def split_dataset( data):
    train, test = data[1:-328], data[-328:-6]
    train = array(split(train, len(train)/7))
    test = array(split(test, len(test)/7))
    return train, test
```

To_series:

Este segundo método nos crea una lista que contiene la primera columna de cada semana en el conjunto de datos. Como hemos mencionado anteriormente, esta columna en nuestro conjunto de datos hace referencia al parámetro de “datetime”. Las extrae del argumento “data” que es una matriz multidimensional para crear una lista de series temporales correspondientes a cada semana.

La siguiente línea de código se encarga de convertir la lista de series “semanales”, mencionadas en el método anterior, en un array, gracias al método “flatten()”, que nos permite “aplanar” la matriz en una serie unidimensional para poder trabajar con ello.

```
def to_series( data):
    series = [week[:, 0] for week in data]
    series = array(series).flatten()
    return series
```

Evaluate_forecasts:

En nuestro siguiente método, vamos a realizar el proceso para evaluar las predicciones obtenidas en nuestro modelo, en términos de la raíz cuadrada del error cuadrático medio (RMSE) y del error cuadrático medio (MSE). En definitiva, el encargado de indicarnos la calidad de las predicciones de nuestro modelo. En cuanto al código, vamos a indagar paso por paso:

En primer lugar, nuestro método toma dos parámetros, dos matrices con los valores reales y los pronosticados por nuestra red neuronal. Seguidamente, creamos una lista vacía para guardar los valores del RMSE de cada serie temporal pronosticada.

A continuación, guardamos en la variable "mse" el error cuadrático medio de los valores reales de la columna "i" y los valores pronosticados de la columna "i". Posteriormente, hacemos la raíz cuadrada del valor ya calculado para obtener el RMSE y por último lo guardamos en nuestra lista "scores". Este bucle realiza esta operación para todas las columnas de valores pronosticados.

Nuestro segundo bucle "for", se encarga de realizar el error cuadrático medio, pero en esta ocasión para todas las columnas y filas de la matriz. Nos apoyamos en las variables "col" y "row" para acceder a cada valor de los reales y pronosticados. Por último, hacemos la raíz cuadrada como en el bucle anterior para conseguir la raíz cuadrada del error cuadrático medio.

Devolvemos el valor global RMSE y la lista de valores RMSE para cada serie de tiempo pronosticada.

```
def evaluate_forecasts( actual, predicted):
    scores = list()
    for i in range(actual.shape[1]):
        mse = mean_squared_error(actual[:, i], predicted[:, i])
        rmse = sqrt(mse)
        scores.append(rmse)
        s = 0
    for row in range(actual.shape[0]):
        for col in range(actual.shape[1]):
            s += (actual[row, col] - predicted[row, col])**2
    score = sqrt(s / (actual.shape[0] * actual.shape[1]))
    return score, scores
```

Evaluate_model:

La siguiente función es la plantilla básica para evaluar el rendimiento de nuestro modelo. Disponemos del argumento “model_func” encargado de tomar una lista de valores como entrada y devolver las predicciones de la serie futura. También disponemos del arugmento “train”, secuencia de datos de entrenamiento y”test”, secuencia de datos de prueba.

Primero copiamos en “history” la lista de valores de “train” para posteriormente iterar en cada elemento de “test” y realizar una predicción con “model_func” sobre el siguiente elemento. Añadimos este pronóstico a la lista “predictions” y el elemento actual de “test” a “history”.

Por último, utilizamos la ya mencionada función “evaluate_forescast” para obtener las puntuaciones tanto general del modelo como para cada paso de tiempo en los datos de prueba.

```
def evaluate_model( model_func, train, test):
    history = [x for x in train]
    predictions = list()
    for i in range(len(test)):
        yhat_sequence = model_func(history)
        predictions.append(yhat_sequence)
        history.append(test[i, :])
    predictions = array(predictions)
    score, scores = evaluate_forecasts(test[:, :, 0], predictions)
    return score, scores
```

Arima_forecast:

En nuestro último método es donde declaramos nuestro modelo, para este caso está declarado el modelo clásico ARIMA. Este método tiene como parámetro la ya vista lista de valores “history”, con el fin de devolver una predicción de una serie temporal futura, utilizando ARIMA.

Se crea un modelo “ARIMA” con el orden (7,0,0), el primer argumento es la serie de tiempo, obtenida de la llamada al método “to_series” y el siguiente argumento es el orden del modelo. En este caso significa que es un modelo autorregresivo (AR) de orden 7 y sin términos de medias móviles.

A continuación, se ajusta el modelo creado para obtener una predicción para los próximos 7 días, una semana de tiempo, utilizando el método “predict()”, con los argumentos “len(series)” y “len(series)+6”, lo que produce el pronóstico de la serie de tiempo para los siguientes 7 días.

Finalmente devolvemos la predicción.

```
def arima_forecast( history):
    series = to_series(history)
    model = sm.tsa.arima.ARIMA(series, order=(7,0,0))
    model_fit = model.fit()
    yhat = model_fit.predict(len(series), len(series)+6)
    return yhat
```

Resultados:

Tras haber explicado e indagado en primera instancia, en todos los modelos clásicos que vamos a tratar en esta primera parte del documento; y posteriormente revisar el código en Python encargado de realizar esa teoría previa, vamos a comparar y comentar los primeros resultados obtenidos para cada uno de los modelos.

AR

El primer modelo en revisar va a ser el Autorregresivo, como comentamos en el apartado anterior, la implementación en Python utilizando la librería "statsmodels" y apoyándonos en el método "arima()" sería la siguiente:

```
model = sm.tsa.arima.ARIMA(series, order=(7,0,0))
```

A continuación, vamos a hablar de los parámetros del modelo, en este caso recibe 2:

-Series: es la serie de tiempo a analizar, es decir, nuestra variable dependiente a analizar.

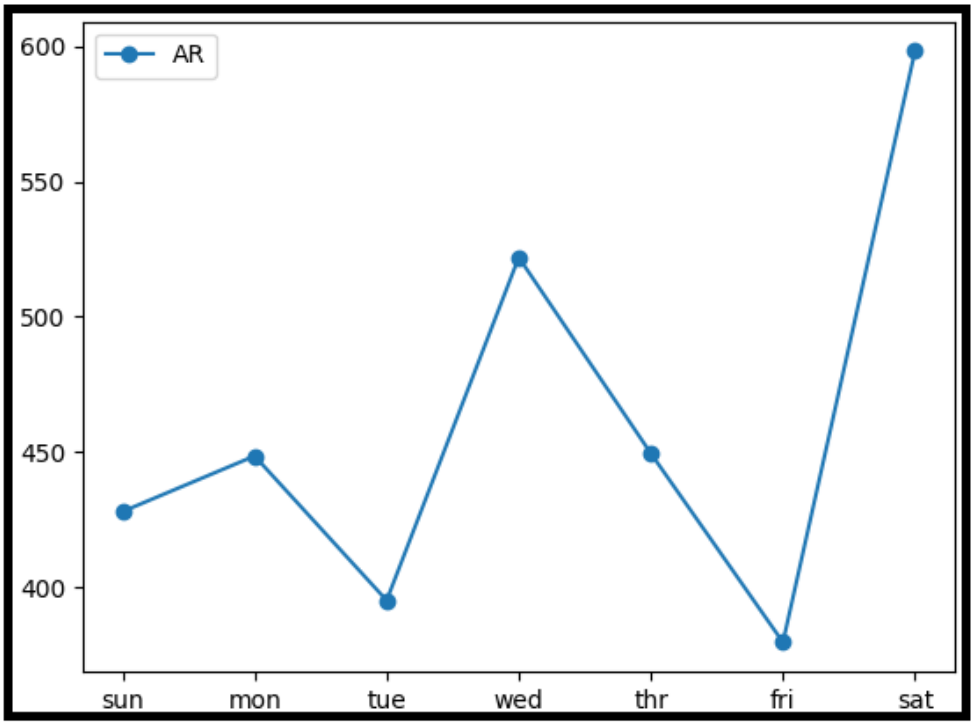
-Order: tupla que contiene los valores (p,d,q) de nuestro modelo. Vamos a explicar en detalle el significado de los tres parámetros ya que serán recurrentes a lo largo de esta comparativa, y es necesario tenerlos claros.

1)P(Orden autorregresivo): este parámetro nos indica el número de valores anteriores de la variable dependiente que vamos a utilizar para predecir nuestro valor actual. En definitiva, si p=2 significa que utilizamos los dos valores anteriores para realizar la predicción actual.

2)D(Orden de diferenciación): representa el número de veces que se debe diferenciar una serie temporal para obtener una serie estacionaria. A modo de recordatorio, una serie estacionaria es aquella donde la media y la varianza son constantes a lo largo del tiempo.

3)Q(Orden de la media móvil): indica el número de errores pasados que utilizamos para predecir el valor actual. Similar al parámetro "P", en este caso, q=1 significa que se utiliza el error de la predicción anterior para hacer la predicción actual.

Una vez comentado esto vamos con nuestros primeros resultados. A lo largo de esta comparativa vamos a recurrir a una gráfica, s en el eje vertical se van a representar los valores de los RMSE en base a los días de la semana, que vemos en el eje horizontal. Así mismo, debajo de cada gráfica, vemos el RMSE global y el RMSE de cada día de la semana pronosticado, que es el representado anteriormente.



Resultados modelo AR

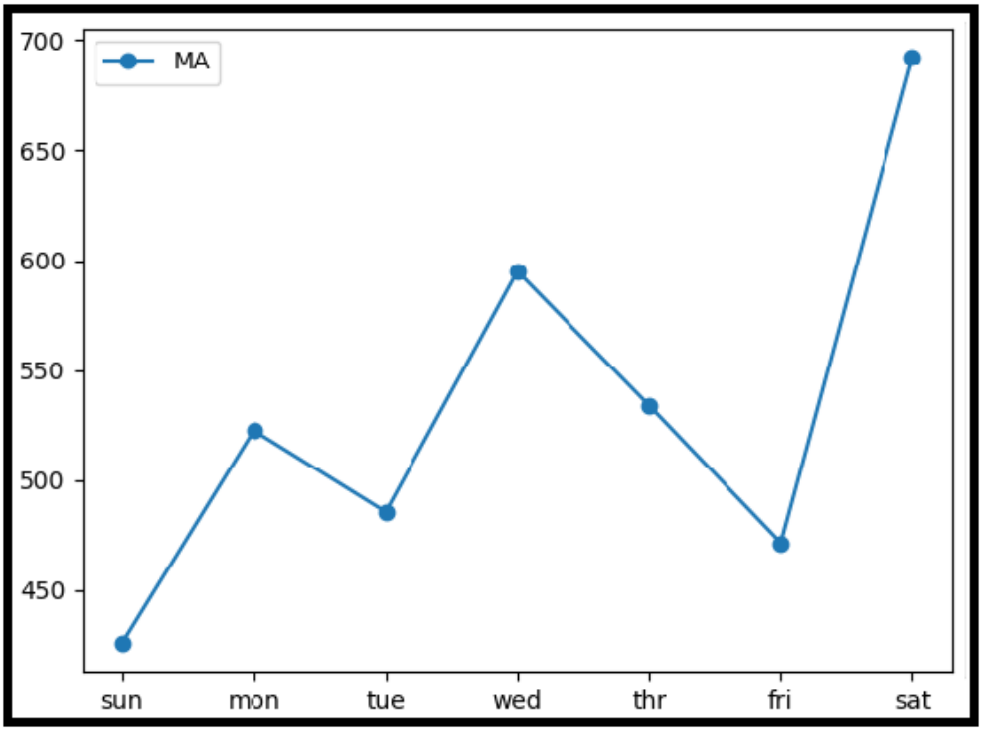
```
AR: [465.476] 427.9, 448.5, 395.0, 521.9, 449.4, 379.6, 598.3
```

Hemos elegido el valor 7 para este primer acercamiento, debido a la estacionalidad semanal de la serie

MA

En segundo lugar, vamos a utilizar el modelo de la media móvil (MA). Para este caso, volvemos a utilizar el método ARIMA de la librería "statsmodels", aunque en esta ocasión variamos los parámetros que recibe. Como ya mencionamos, solo vamos a utiliza el valor "q" dentro del parámetro *order* para referirnos al modelo MA.

```
model = sm.tsa.arima.ARIMA(series, order=(0,0,1))
```



Resultados modelo MA

```
MA: [538.627] 425.9, 522.7, 485.7, 594.9, 534.0, 471.7, 692.0
```

A primera vista, vemos que el RMSE global es superior al del modelo anterior AR, esto quiere decir que los datos tienen una autocorrelación fuerte. En otras palabras, este modelo se basa en el análisis de los errores previos de una serie temporal para predecir el actual, mientras que el AR se fundamenta en los valores previos; esto indica que la serie tiene una dinámica con patrones de autocorrelación, como tendencias o patrones estacionarios.

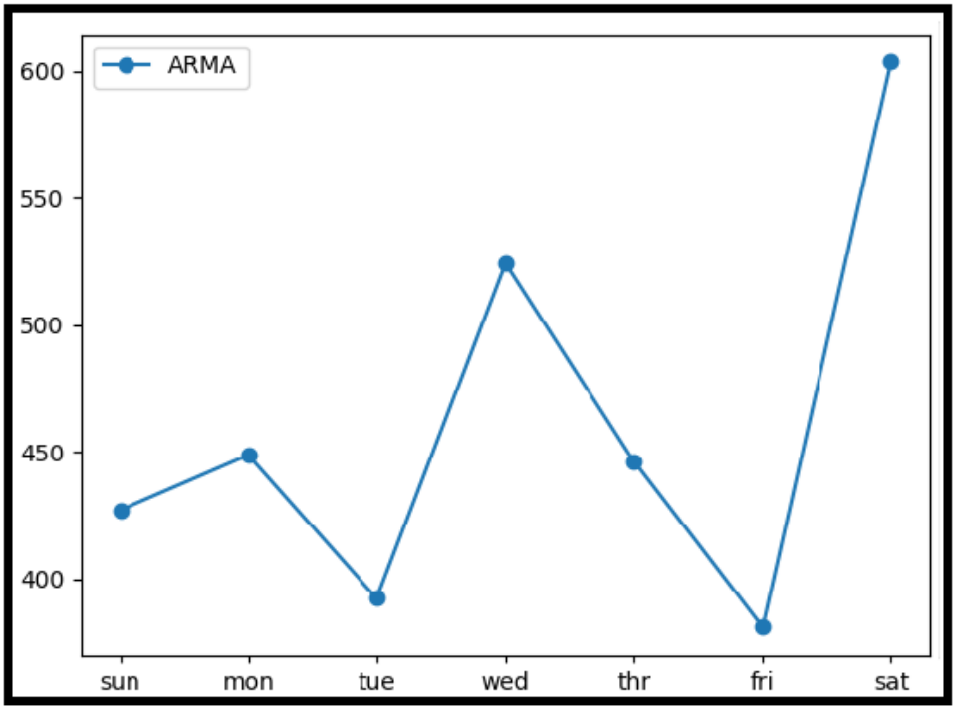
Como apunte adicional, se probó a aumentar el valor "q" para tener en cuenta más errores previos. Con esto se consiguió una mejora en los resultados, pero quedan distantes de los obtenidos por el modelo autorregresivo.

ARMA

Nuestra siguiente modelo es el ARMA, que como recordamos es la combinación de los dos anteriores, el modelo autorregresivo (AR) y el de la media móvil (MA).

```
model = sm.tsa.arima.ARIMA(series, order=(7,0,1))
```

Volvemos a utilizar el método "arima()", en este caso, para que los resultados sean lo más homogéneos y comparables posibles, hemos optado por respetar los parámetros *utilizados anteriormente para los modelos AR y MA, siendo "p=7" y "q=1".*



Resultados modelo ARMA

```
ARMA: [466.236] 427.1, 448.8, 392.5, 524.2, 446.5, 381.2, 603.6
```

Los resultados obtenidos son casi idénticos a los aportados por el modelo AR, 465.476. Esto nos podría indicar que la estructura de los datos, como habíamos supuesto en el apartado anterior, tiene una fuerte autocorrelación entre los valores o alguna clara tendencia entre los mismo. Con lo cual, con respecto a la parte del error móvil (MA), vemos que es posible que sea cierto lo afirmado anteriormente que los valores no son muy dispares, con lo que el error móvil no da tan bueno resultados.

Por último, hicimos un reajuste en los órdenes de la función “arima()” para tener en cuenta más errores previos, como vimos anteriormente, esto nos ayudó a mejorar un poco. En este caso se quedaría así:

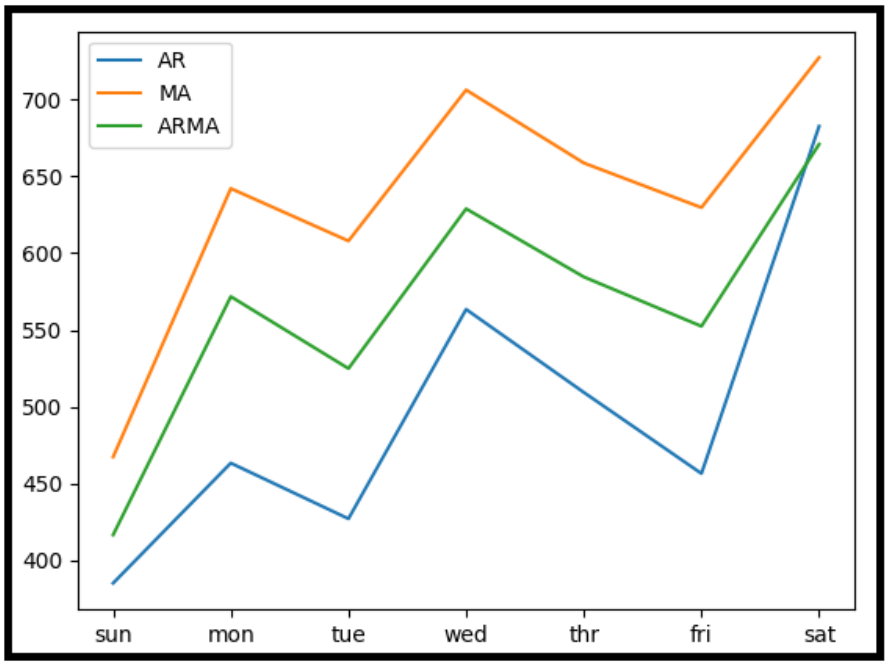
```
model = sm.tsa.arima.ARIMA(series, order=(7,0,3))
```

```
ARMA: [454.041] 394.4, 435.7, 391.9, 527.3, 433.3, 359.2, 591.1
```

Conseguimos reducir un poco los valores, aunque seguimos observando como el peso de la parte relacionada con la MA, no es tan determinante para este conjunto de datos.

Comparativa modelos AR MA y ARMA:

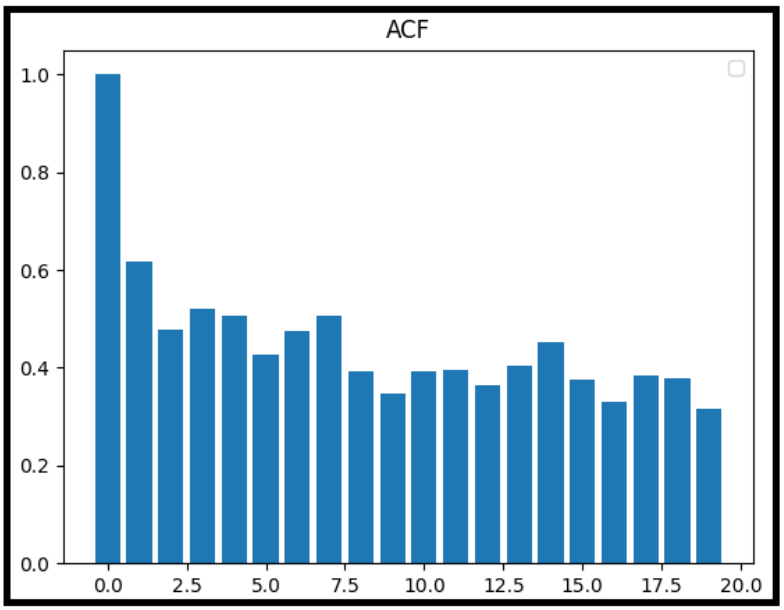
Tras haber repasado los tres modelos por separado, vamos a confrontarlos y así sacar conclusiones de cual funciona mejor o peor y el motivo detrás de ese comportamiento. En primer lugar vamos a echar un vistazo a las versiones más sencillas de estos tres modelos, AR(1), MA(1) y ARMA(1,1):



Comparativa AR MA ARMA

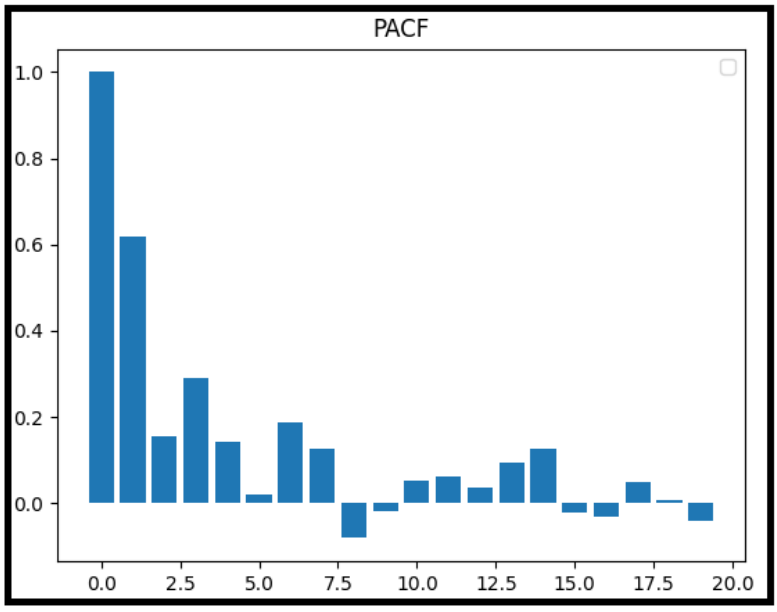
En un primer vistazo, observamos lo siguiente: el modelo AR es el que mejores resultados da en su variante más sencilla, mientras que el MA obtiene resultados mucho peores, y el ARMA al ser la combinación lineal de ambos modelos, se queda en un término intermedio. La primera pregunta que tenemos que responder es si es un comportamiento esperable en base a nuestro conjunto de datos. Para ello vamos a realizar un estudio de las componentes ACF Y PACF, que son las desencadenantes de que nuestro modelo MA y AR respectivamente encajen y arrojen mejores resultados en el modelo planteado.

Hemos realizado un análisis de ambas y vamos a revisarlas para ver con que valores deberíamos ajustar nuestro modelo para que nos brinde las mejores predicciones:



ACF

Basado en lo que nos arroja la ACF, deberíamos empezar con el proceso MA(1), ya que vemos que el primer "lag", también llamado retardo de nuestra serie temporal es el más significativo de la misma, con lo que vamos a utilizar el parámetro 1 de la media móvil, que implicará que solo utilizará 1 error previo para su cálculo.



PACF

En esta gráfica, vemos lo comentado en el análisis del conjunto de datos. Donde comentábamos que los mejores valores para empezar, por ejemplo, pueden ser el AR(1), AR(3), AR(6) y AR(7). En base a los estudios y pruebas previas, concluimos que el mejor modelo AR que está funcionando para nuestros datos es el AR(7), no obstante, la diferencia no es circunstancial.

ARIMA

ARIMA, será el siguiente en ser utilizado para realizar el pronóstico de nuestra serie temporal. Como en todos los anteriores utilizaremos el método “arima()” de la libería “statsmodels”

```
model = sm.tsa.arima.ARIMA(series, order=(7,1,1))
```

Como mencionamos para el modelo ARMA, con el fin de intentar observar la influencia en este caso de la integración del modelo hemos seguido con los mismos valores para “p” y “q”, que eran 7 y 1, ,. La integración,aportada por el orden “d”, el cual toma el valor 1 en esta primera medición.

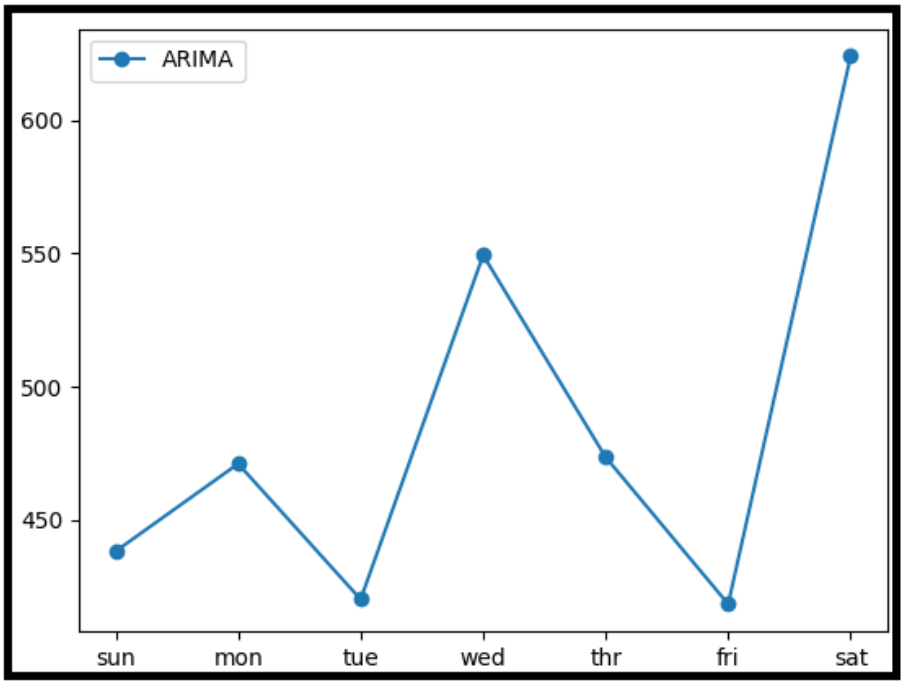


Ilustración resultados ARIMA

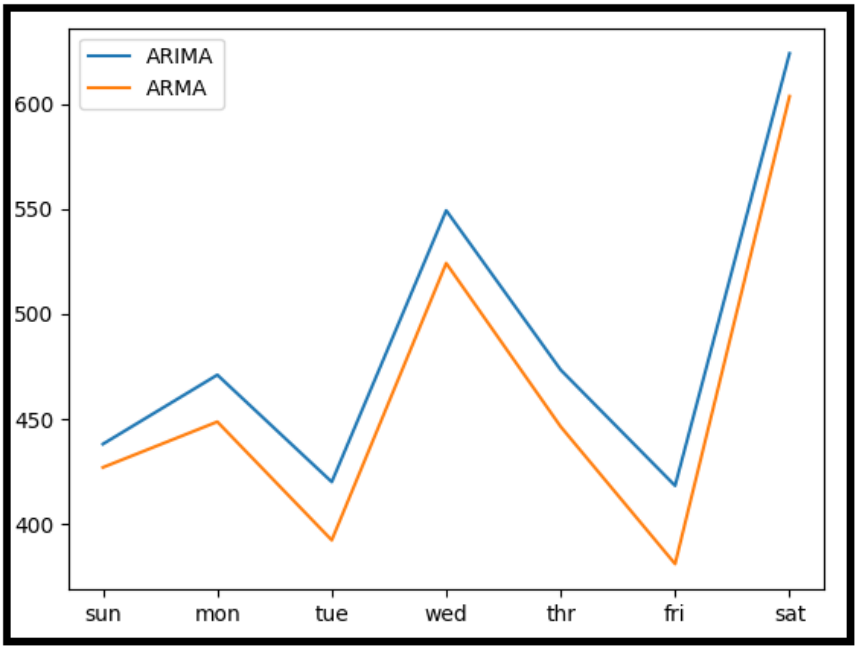
```
ARIMA: [490.039] 438.2, 471.1, 420.2, 549.3, 473.5, 418.4, 624.0
```

Vemos resultados inferiores a los anteriores modelos AR y ARMA, esto indica que la serie temporal ya es estacionaria y no necesita una diferenciación para poder obtener propiedades estadísticas estacionarias. Tiene sentido pensar en nuestra serie temporal como algo estacionario debido a la naturaleza de los datos, donde se tratan de semanas cíclicas donde se observa el consumo energético de un hogar.

Si por ejemplo se tratase de una serie temporal que muestra las ventas mensuales de un producto a lo largo de un mes, y esta tiene implícita una tendencia accedente, puede ser difícil para el modelo ARMA,y ese caso la diferenciación del modelo ARIMA facilitaría el pronóstico.

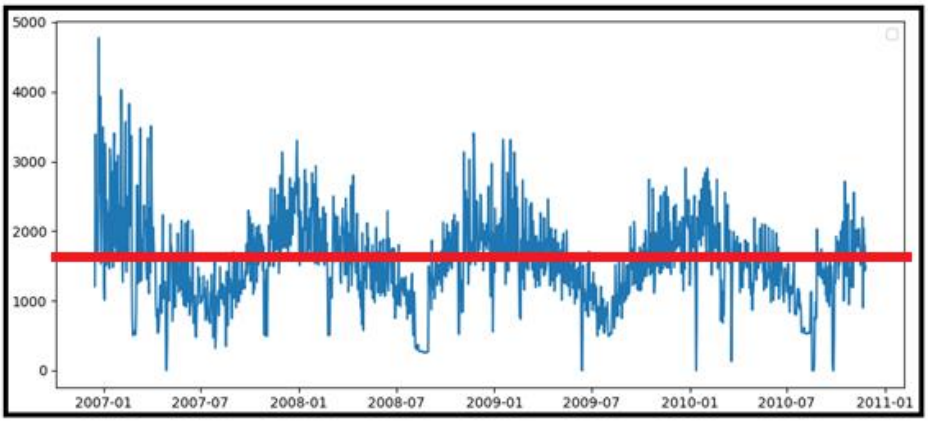
Comparación con ARMA:

Antes de continuar, como en el modelo anterior, vamos a realizar una comparación con ARMA y ver si los resultados que nos arrojan tienen sentido y son coherentes con el conjunto de datos a predecir.



Comparación ARMA ARIMA

Como se puede apreciar, no hay que ahondar mucho para poder afirmar lo que veníamos comentando en puntos anteriores, que se trata de una serie estacionaria, lo que quiere decir que tiene una media que tiende a ser constante. Lo podemos observar en la siguiente gráfica:



gráfica de la media

Con lo que la integración no dará mejores resultados, ya que esta nos ayuda a transformar una serie temporal con una tendencia no constante en el tiempo a otra donde sea constante, con el fin de que nuestros modelos AR, MA y ARMA funcionen de la mejor manera. Como hemos comentado, esto no ocurre con lo que nuestros resultados son similares, con integración o sin integración.

Por último, al igual que en modelo anterior, vamos a jugar con el parámetro relativo a la media móvil (MA) para ver si obtenemos mejores resultados:

```
model = sm.tsa.arima.ARIMA(series, order=(7,1,3))
```

```
ARIMA: [454.563] 396.2, 432.9, 392.6, 527.8, 436.9, 365.4, 587.3
```

Como pasaba en el caso anterior, nuestro modelo rinde mejor teniendo en cuenta los tres errores anteriores en lugar de solamente uno. El valor obtenido es casi idéntico al presentado por el modelo ARMA para esta misma configuración, con lo que, volvemos al mismo punto donde creemos que la diferenciación aquí no juega un papel fundamental.

SARIMA

Nuestra última predicción será realizada ayudados del modelo SARIMA, que como su nombre indica es el ya comentado modelo ARIMA, pero añadiendo la S de estacionalidad. Este modelo ayuda a tener en cuenta las posibles estacionalidades de nuestro conjunto de datos, por ejemplo, en nuestro caso suponemos que tenemos una estacionalidad semanal en el consumo energético de cada hogar.

```
model = sm.tsa.SARIMAX(series, order=(7,1,1),seasonal_order=(1, 0, 0, 12))
```

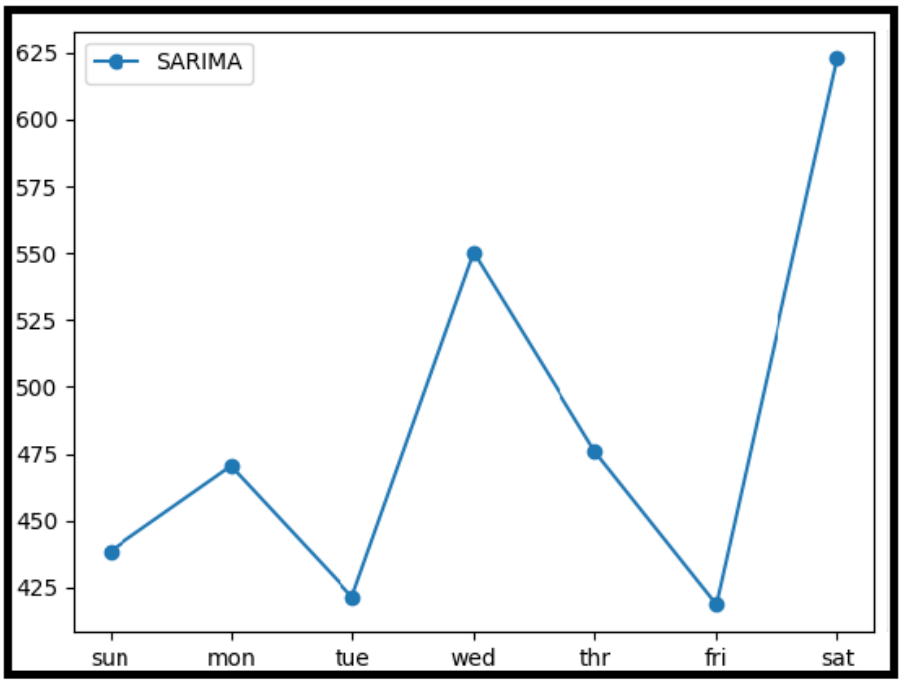
Para este caso, tenemos que cambiar la ya utilizada función “arima()” para utilizar “SARIMAX” que es la encargada de implementar el modelo SARIMA en Python. En cuanto a los parámetros, ahora recibe 3, los dos primeros son exactamente iguales a los aportados para el método “arima()”, sin embargo ahora disponemos de un “seasonal_order” que tiene los siguientes parámetros:

-P (Orden autorregresivo estacional): este valor hace referencia al SAR, que modela la dependencia de la serie temporal en cuanto a valores previos del mismo periodo estacional.

-D (Grado de diferenciación estacional): el SI viene a referenciar el número de veces que se debe diferenciar la serie temporal en el periodo de estacionalidad para hacerla estacionaria.

-Q (Término de media móvil estacional): el SMA modela la dependencia de la serie temporal en errores previos del mismo periodo estacional.

-S: Nos indica cuánto dura el periodo de estacionalidad.



Resultados SARIMA

```
SARIMA: [490.429] 438.4, 470.4, 421.4, 550.4, 475.9, 418.5, 622.9
```

En primera instancia, vemos unos resultados algo inferiores a los contemplados en el modelo anterior, el cual es la variante sin la estacionalidad. No obstante, como hemos realizado con los modelos anteriores, ajustando los parámetros de la función "sarimax()" de statsmodels, podemos obtener mejor resultados.

```
model = sm.tsa.SARIMAX(series, order=(3,1,3),seasonal_order=(1, 1, 1, 7))
```

Hemos modificado la primera parte, para acomodarnos a los cambios realizados anteriormente, ajustando el término autorregresivo y la media móvil a 3 y 3 respectivamente, ya que hemos comprobado que para nuestra serie temporal dan mejores resultados. Y lo podemos comprobar a continuación:

```
SARIMA: [468.227] 422.6, 464.2, 405.4, 531.8, 445.4, 379.9, 592.2
```

Tras la modificación realizada en los parámetros relativos al parámetro *order*, vemos como bajamos casi al mismo resultado que nuestro anterior modelo, no obstante seguimos obteniendo peores resultado que los vistos anteriormente. Esto se puede deber a que el nuevo parámetro "*seasonal_order*" relativo a la capacidad estacional del modelo, no está correctamente ajustado. En consecuencia, para este último entrenamiento vamos a modificarlo con el fin de estudiar su comportamiento, conociendo la estacionalidad de los datos a predecir.

```
model = sm.tsa.SARIMAX(series, order=(3,1,3),seasonal_order=(3, 0, 1, 7))
```

Para conseguir mejorar el resultado, hemos realizado modificaciones en la parte estacional, como veníamos anticipando. Subiendo el parámetro relativo al SAR

(*Seasonal Autoregresive*) que es el más influyente para este conjunto de datos, como se ha venido observando en base a los resultados obtenidos.

```
SARIMA: [456.370] 407.4, 433.5, 395.5, 526.0, 439.7, 369.9, 583.6
```

Antes de llegar a este resultado, se probó con la misma configuración de datos que para la variable "*order*", ignorando el parámetro estacional, sería -> 3,1,3. Para esta configuración los resultados empeoraron significativamente, con que pasamos a rebajar el parámetro relativo a la media móvil y eliminar el parámetro de integración, con esto hemos conseguido nuestro mejor resultado para este modelo, 456.370.

Conclusiones:

Tras realizar un estudio de todos los modelos clásicos, en esta aplicación para nuestro conjunto de datos que trata el consumo energético en una casa, podemos sacar las siguientes conclusiones:

Los modelos clásicos funcionan bien en nuestro conjunto de datos debido a la naturaleza de este. Son estacionarios y tienen una clara dependencia de unos valores de consumo con respecto a consumos anteriores. El modelo AR funciona mejor que el modelo MA por la fuerte dependencia lineal que existe entre el valor o conjunto de valores anteriores respecto al actual, mientras que el error por el que intenta pronosticar el modelo MA no es tan efectivo en este caso.

Posteriormente, estudiamos la combinación de ambos modelos, ARMA, el cual nos arrojó un resultado proporcional a su nombre, uno a medio camino este el modelo autorregresivo y el relativo a la media móvil.

Hablando ya de la última parte de estos modelos clásicos, tenemos ARIMA y SARIMA. En primer lugar, ARIMA, no tiene gran aportación en nuestra serie, nuevamente debido a la estacionalidad de los valores a lo largo deltiempo, no teniendo una tendencia ascendente o descendente a medida que avanza la serie temporal. Con lo que, a efectos prácticos, sin integración nuestro modelo ARIMA se comporta como un modelo ARMA.

Finalmente, el modelo SARIMA, que es la versión de ARIMA que intenta tener en cuenta el comportamiento estacional de nuestra serie, en este caso anual como se ha visto en gráficas anteriores. Este modelo es el que mejores datos arroja, y permite concluir que nuestro estudio sobre el conjunto de datos de consumo energético es acertado.

Ahora compararemos estos resultados obtenidos con modelos basados en redes neuronales. En el siguiente punto, seguiremos el mismo esquema que hemos utilizado hasta ahora, presentaremos los modelos a utilizar, los estudiaremos, veremos su implementación y su desempeño con nuestro conjunto de datos para posteriormente compararlos entre ellos y con sus antecesores, los modelos clásicos.

Aprendizaje profundo (Deep Learning):

Tras el estudio realizado en los modelos clásicos, vamos a continuar con los modelos basados en redes neuronales. Cabe destacar que nuestro principal objetivo es intentar evidenciar que los métodos basados en "Deep Learning" no desempeñan un mejor rendimiento que los modelos clásicos a la hora de predecir series temporales.

Multi Layer Perceptron (MLP):

El modelo MLP o perceptrón multicapa es una de las redes neuronales más básicas y su estructura puede reducirse a lo mínimo, una capa de entrada, una capa oculta y una última capa de salida. Aunque a simple vista es una forma de red simple en comparación con otras que se estudiarán más adelante como las CNN o RNN, estas son eficaces a la hora de aprender y modelar relaciones no lineales complejas en los datos, que al final es lo que estamos tratando en nuestro conjunto de datos.

Otro problema al que nos enfrentamos con este tipo de modelos es que por sí solas no son capaces de tener en cuenta la temporalidad de los datos a menos que se les proporcione de alguna forma.

Univariate:

El primer modelo de perceptrón multicapa que vamos a estudiar es el "Univariate", que viene a indicar que solo tomará una variable como entrada para obtener la predicción, similar al comportamiento que veníamos tratando en los modelos clásicos. Supongamos que tenemos una serie temporal que recoge la temperatura diaria en un pueblo a lo largo de varias semanas, utilizamos los últimos 7 días para realizar la predicción del siguiente, en otras palabras, tomamos 7 "*lags*" para predecir el siguiente instante en nuestra serie temporal, este sería el funcionamiento que vamos a seguir.

Con esto que acabamos de comentar, y volviendo a lo visto en la introducción del MLP, hemos sido capaces de "pasarle" la temporalidad al modelo. Sin embargo, esta temporalidad es muy simple y no entiende los posibles patrones que puede tener nuestra serie a la hora de realizar su estudio. Únicamente tiene en cuenta la "temporalidad" de los datos más allá del número de pasos de tiempo o "lags" que estemos usando, lo cual evidencia que conceptos vistos anteriormente como estacionalidad o tendencias a largo plazo de los datos, no podrán ser captadas por este modelo.

Implementación en Python:

Como hicimos con los modelos clásicos, vamos a tratar un pequeño ejemplo en donde comentaremos la manera de preparar nuestro "dataset" y cómo se comporta este modelo de aprendizaje profundo.

Para ello, vamos a abstraer nuestra serie temporal al ejemplo más básico, una secuencia de número, donde tendremos que predecir el siguiente, por ejemplo:

[10, 20, 30] -> 40

En nuestro caso, será una secuencia del 10 al 90, con el fin de que sea capaz de pronosticar que el siguiente valor es el 100. Para ello se ha implementado el siguiente código en Python:

```
def split_sequence(sequence, n_steps):
    X, y = list(), list()
    for i in range(len(sequence)):
        end_ix = i + n_steps
        if end_ix > len(sequence)-1:
            break
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)
```

Nuestro primer método en el código será "Split_sequence()", que como su nombre indica, se encarga de dividir una secuencia de entrada en ventanas de tiempo.

En primer lugar, el método toma como entrada una secuencia de datos, "sequence", y un valor "n_steps" que nos indica la longitud de la ventana de tiempo. Se crean dos listas, "X" e "y", que serán devueltas al final del método, con las ventanas de tiempo y las etiquetas correspondientes.

Se itera con un bucle "for" a lo largo de la secuencia, donde nuestro objetivo será hacer esas ventanas de tiempo o "sublistas" guardándolas en la variable "seq_x" para luego añadirlas a la lista "X". El bucle itera hasta que "end_ix", el cual es el valor de la "i" sumado a la longitud de la ventana.

Como resultado devuelve las dos listas. Como ejemplo sería lo siguiente:

Entrada -> sequence = [10,20,30,40,50,60]

n_steps = 2

Salida -> X = [[10, 20],

[20, 30],

[30, 40],

[40, 50]]

y = [30, 40, 50, 60]

Vemos como efectivamente el método crea varias ventanas de tiempo de longitud 2 y sus correspondientes “etiquetas”, que son los elementos inmediatamente posteriores a esa ventana de tiempo.

```
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=2000, verbose=0)
```

En segundo lugar, tenemos nuestro modelo, que será creado utilizando la librería “*Keras*” para definirlo.

En la primera línea, creamos un modelo secuencialque nos permite aplicar capas de manera secuencial, como su nombre indica. Seguidamente, añadimos una capa “Dense” con 100 neuronas (primer parámetro), función de activación “relu” y por último especificamos la dimensión de los datos de entrada, que será “n_steps”.

Como aclaración, la función de activación “relu”, utilizada en este modelo, realiza la transformación de cualquier valor negativo a 0, mientras mantiene los valores positivos inalterados.

Ahora, en la tercera línea, añadimos otra capa “Dense”, pero esta sin parámetros, con una única neurona. Esto se realiza de esta forma ya que es la capa de salida de nuestro modelo. Al no utilizar ninguna función de activación, esta utiliza la “lineal”, la cual deja sin cambios a los valores que pasen por ella.

$$y = w_0 + w_1*x_1 + w_2*x_2 + w_3*x_3 + ...$$

Ejemplo de función lineal

Finalmente, tenemos la línea donde se compila el modelo. Para ello se indican por parámetros el optimizador y la función de pérdida. Como función de pérdida tenemos el ya conocido error cuadrático medio (MSE), que habíamos utilizado y explicado en la implementación de los modelos clásicos. Por otra parte, el optimizador, es aquel encargado de actualizar el peso de las neuronas durante el proceso de entrenamiento, en este caso utilizamos “Adam” (Adaptive Moment Estimation). Este optimizador, cabe destacar que es uno de los optimizadores basados en gradiente más utilizados hoy en día.

Por último, igual que en los modelos clásicos, entrenamos la red neuronal. En este caso, le pasamos como parámetros las listas “X” e “y”, correspondientes a las ventanas de tiempo y sus respectivas etiquetas. El número de épocas, 2000, el número de

“pasadas” que se realizarán por el modelo y “verbose=0”, que simplemente indica que no se imprimirá por consola nada sobre el entrenamiento de la red.

Ahora pasemos a ver los resultados obtenidos para la secuencia [10, …., 90], en este caso, como hemos comentado deberíamos obtener el valor 100, ya que los valores distan 10 unidades entre sí y todos son consecutivos.

```
[[100.61352]]
```

Efectivamente, obtenemos un valor muy próximo a 100 lo que nos indica que nuestra red ha entrenado correctamente.

conjunto de datos:

En el siguiente apartado vamos a realizar las comprobaciones de nuestro modelo con respecto al dataset planteado en los modelos clásicos, el consumo energético en el hogar. Para este punto seguimos utilizando “Global_active_power” como variable a predecir y como la única en la que nos apoyamos para realizar las predicciones. En otras palabras, en esta primera puerta de entrada a la comparación entre modelos, seguimos sin utilizar el resto de “variables” o “features” del conjunto de datos, al igual que pasaba en los ya estudiados modelos clásicos.

En primer lugar, se han realizado entrenamientos de varios modelos distintos, con diversas configuraciones para intentar obtener el mayor beneficio de este en esta prueba. Sin embargo, el objetivo de nuestro trabajo de fin de grado es comparar este con los modelos clásicos para ver si efectivamente nos arrojan mejores resultados, con lo que en este apartado no se va a realizar un análisis exhaustivo para determinar si el modelo seleccionado con la distribución elegida es la óptima para este problema que nos acontece.

Vemos como para este primer entrenamiento, nuestro modelo se compone de varias capas densas, con 200, 100 y 500 neuronas respectivamente, y la última cuenta con un “kernel_regularizer”. La cual, lo único que añade al modelo es una penalización a la función de pérdida proporcional a la magnitud cuadrada de los pesos del modelo.

Implementación:

```
def keras_forecast(train,test,n_steps):
    train_x, test_x = split_dataset(train)
    train_y, test_y = split_dataset(test)
    model = Sequential()
    model.add(Dense(200, activation='relu',input_dim=n_steps))
    model.add(Dense(100,activation='relu'))
    model.add(Dense(500,activation='relu', kernel_regularizer=l2(0.01)))
    model.add(Dense(1))

    model.compile(optimizer='adam', loss='mse')

    model.fit(train_x, train_y, epochs=100, verbose=1)
```

Épocas -> 100.

```
MLP: [424.4] 472.5, 454.3, 482.9, 491.8, 519.5, 404.2, 328.2
```

Épocas -> 500.

```
MLP: [406.0] 461.0, 436.0, 423.2, 515.7, 396.3, 357.4, 376.8
```

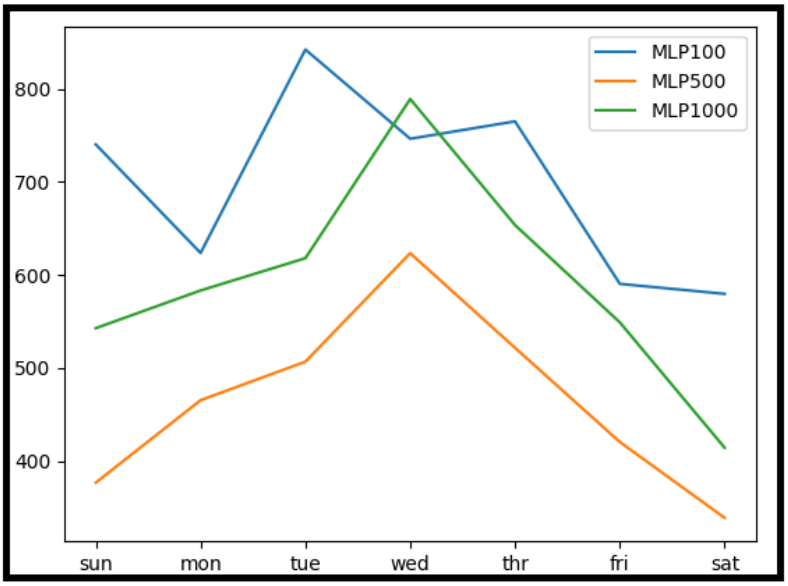
Épocas -> 1000.

```
MLP: [542.6] 542.7, 583.2, 617.9, 788.9, 653.5, 548.9, 414.2
```

Como podemos ver, hay una mejora significativa en función del número de épocas que entrene la red, aunque al llegar a las 1000 épocas vemos como el modelo presenta un deterioro. Este apartado es otro que vamos a tratar en esta parte del proyecto, el sobre entrenamiento u “overfitting”, esto quiere decir que nuestra red se está sobre ajustando.

Nos referimos a que nuestro modelo se está ajustando en exceso a los valores de entrenamiento y tiene dificultades para generalizar los nuevos datos. Podemos ver esto como un estudiante que a la hora de prepararse un examen no utiliza el temario, sino que memoriza todas las preguntas que pueden entrar. Si a la hora de la verdad, le toca responder a una pregunta de las estudiadas previamente, no habrá problema, pero si tiene que responder algo ajeno a lo visto, al no tener ningún conocimiento con el cual intentar comprender lo que está leyendo, no podrá hacer nada. Con nuestro modelo pasa un comportamiento similar.

Antes de entrar con la siguiente configuración de nuestro Multi Layer Perceptron, visualizamos una comparación de los tres entrenamientos realizados:



Comparación en las distintas épocas de entrenamiento

La gráfica muestra una comparativa de la media de errores por día en la semana que se producen a la hora de realizar una predicción en función de los distintos modelos. Como se trata de un error, cuanto más próximo esté a cero, indicará que estamos obteniendo mejores resultados, en este caso vemos como el perceptrón "univariate" es aquel que arroja mejores resultados.

Analizaremos un segundo modelo, similar al anterior, cambiando ligeramente la arquitectura de la red neuronal, aumentamos el número de neuronas en cada capa, y observando el número de épocas a partir de las cuales se empieza a producir overfitting para de esta forma poder hacer una parada temprana.

En esta ocasión conseguimos apreciar mejores resultados, no distan en exceso de lo ya observado en modelos anteriores, pero sí obtenemos una pequeña mejoría en nuestro primer acercamiento. Como pasaba antes, y debido a que los modelos son muy parecidos, al entrenar 1000 épocas, nuestro modelo empieza a perder eficacia.

```
model = Sequential()
model.add(Flatten(input_shape=(n_steps, 8)))
model.add(Dense(1000, activation='relu'))
model.add(Dense(700,activation='relu'))
model.add(Dense(500,activation='relu', kernel_regularizer=l2(0.01)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(train_x, train_y, epochs=100, verbose=1)
```

Épocas -> 100.

```
MLP: [525.4] 515.3, 605.0, 663.6, 586.6, 548.3, 417.8, 444.4
```

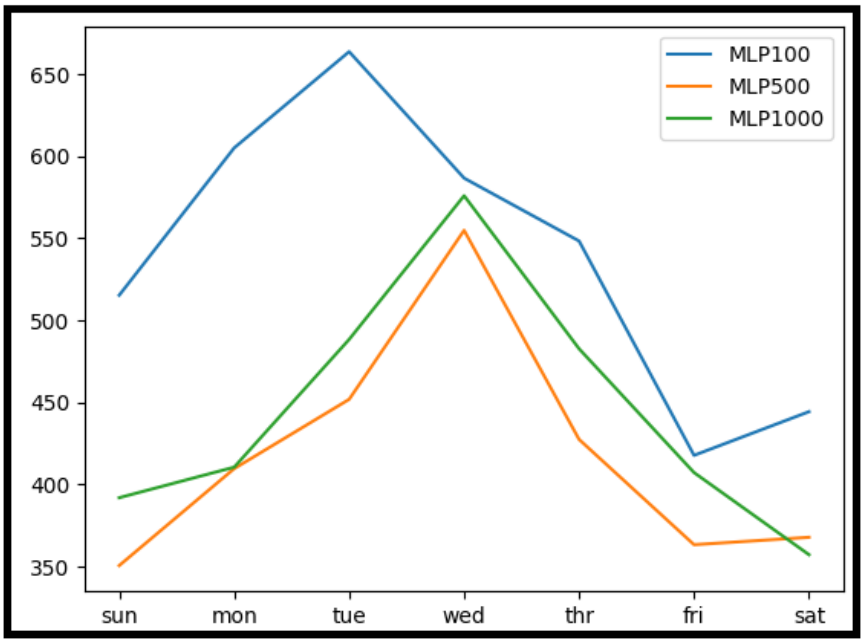
Épocas -> 500.

```
MLP: [400.5] 350.7, 409.7, 451.9, 554.8, 427.5, 363.4, 367.9
```

Épocas -> 1000.

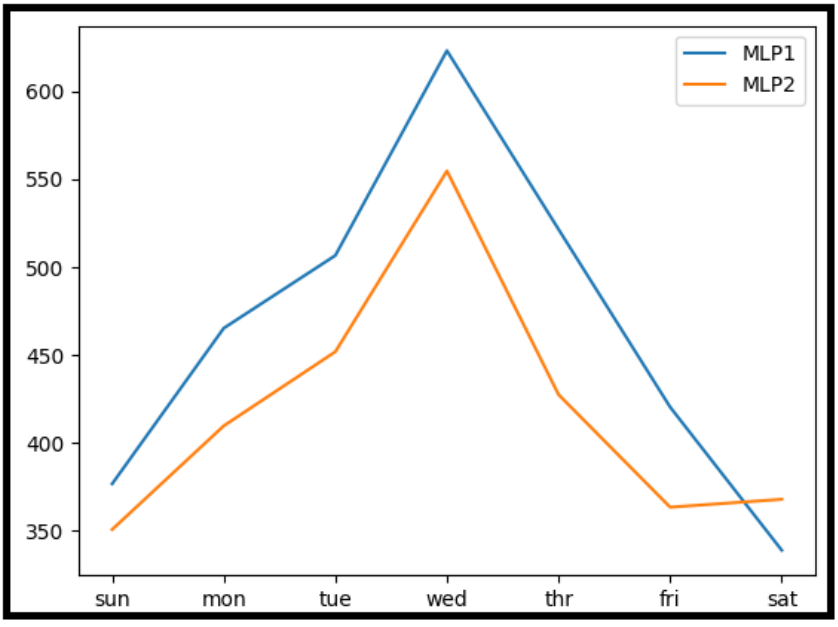
```
MLP: [423.4] 392.0, 410.6, 488.4, 575.9, 482.7, 407.3, 357.3
```

Como hicimos con el modelo anterior, traemos una comparativa para poder examinar con mejor contraste los datos que nos arrojan los resultados presentados:



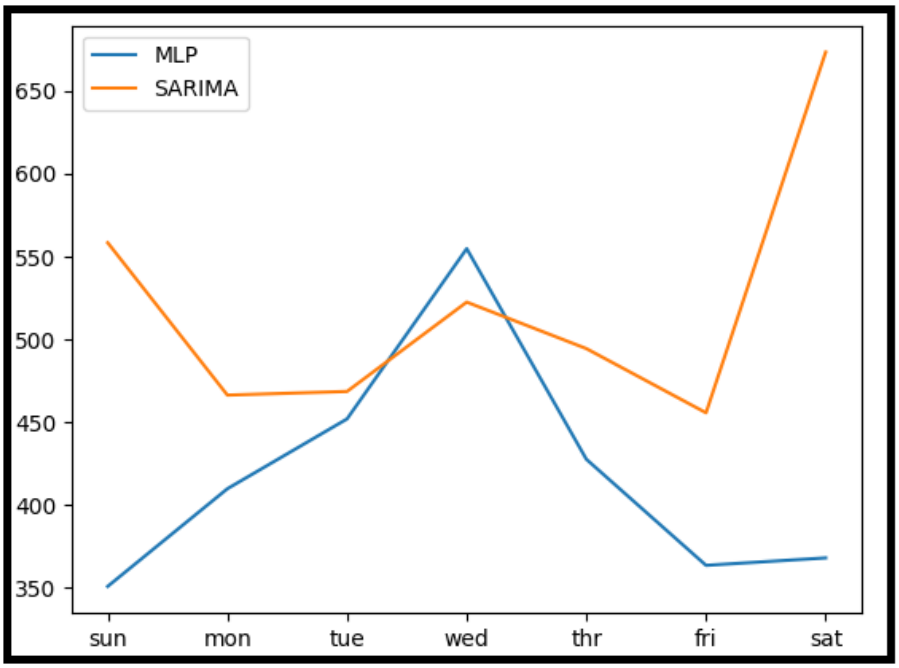
Comparación en las distintas épocas de entrenamiento

Volvemos a observar tendencias similares que evidencian todo lo comentado anteriormente en este primer modelo que estamos tratando en este apartado.



Aquí vemos a los dos mejores respectivamente, los dos de las quinientas épocas de entrenamiento enfrentados en base al error que presentan.

Tras realizar las comparaciones para determinar cuál de las configuraciones era la mejor de este primer apartado del MLP, procedemos a comprarlo con el modelo SARIMA de la parte de los modelos clásicos. Recordemos que SARIMA, es la vertiente "seasonal" de ARIMA, que intentaba captar la posible componente de repetición que experimentara nuestro conjunto de datos.



Comparación resultados MLP SARIMA

Multivariate:

Dentro de este apartado del Multi Layer Perceptron, vamos a ver la segunda variante de este. Venimos de tratar a la versión más básica, univariate, que se caracterizaba por tener como entrada únicamente una variable. Esto tiene sus ventajas a la hora de resolver problemas de regresión o clasificación donde se tiene en cuenta una variable de entrada para predecir la de salida, o donde existen varias, pero únicamente una es aquella que importa a la hora de realizar el análisis.

MLP Multivariate, en nuestra segunda variante tenemos, como el propio nombre indica, múltiples variables de entrada. Esto implica un tratamiento más complejo, que nos va a permitir capturar relaciones más complejas entre las múltiples variables de entrada y la variable de salida de la red. Para nuestro dataset este es un punto de inflexión interesante, ya que es la primera vez donde vamos a ver el comportamiento cuando entran en juego más variables, para ello vamos a realizar como apartado extra un análisis del componente de correlación entre las distintas "features" o atributos.

Implementación en Python:

Antes de entrar en la parte más densa del estudio de nuestro conjunto de datos, vamos a empezar con los modelos más básicos para ver el comportamiento y el funcionamiento cuando disponemos de más de una secuencia de entrada.

A continuación, vamos a tratar la implementación en Python, en este caso, obviaremos todo el código en común con el modelo anterior para centrarnos en las diferencias entre ambas implementaciones.

En primer lugar, vamos a modificar el planteamiento de nuestro problema, y con ello nuestros datos de entrada. Como veníamos comentando, ahora disponemos de dos variables de entrada en lugar de 1, es decir, nuestro paradigma sería el siguiente:

-secuencia 1 -> [10, 20, 30, 40]

-secuencia 2 -> [15, 25, 35, 45]

Con esto buscaríamos obtener el siguiente número en la "secuencia 2" y "secuencia 1" con el fin de sumarlos, para ello observamos que siempre es 10 más que la anterior, o 5 más que su misma posición en la "secuencia 1". Con lo que quiere decir, que, para estas dos variables de entrada, esperaríamos una variable de salida -> 85.

Tras haber comprendido el objetivo de nuestra nueva red neuronal, vamos a proceder con la preparación de los datos:

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])

in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))

dataset = hstack((in_seq1, in_seq2, out_seq))
```

En la imagen anterior, vemos las dos secuencias de entrada ya mencionadas, "in_seq1" e "in_seq2", serán las dos variables que reciba nuestra red. Seguidamente, declaramos la secuencia de salida, "out_seq", que como también adelantábamos en el apartado anterior, es la suma de los dos elementos en las respectivas posiciones de las dos secuencias de entrada.

Posteriormente, ajustamos las dimensiones para poder utilizarlas correctamente en nuestra red neuronal, convertimos los arrays unidimensionales, en una matriz bidimensional con tres filas y una sola columna. Con este cambio realizamos, utilizamos el método "hstack()" de numpy, para apilar las matrices horizontalmente. La variable dataset quería con el siguiente formato:

-dataset -> [[10 15 25]

[20 25 45]

….

[90 95 185]]

Como podemos observar es el resultado de apilar las columnas de nuestras respectivas secuencias.

El siguiente método sería “split_sequences()”, que ha sufrido ligeras modificaciones a la hora de obtener las subsecuencias “seq_x” y “seq_y” como se puede apreciar en la siguiente imagen, esto se debe al cambio en los datos de entrada de la red, pero finalmente, el resultado a devolver es el mismo, una subsecuencia con longitud “n_steps”.

```
def split_sequences(sequences, n_steps):
    X, y = list(), list()
    for i in range(len(sequences)):
        end_ix = i + n_steps
        if end_ix > len(sequences):
            break
        seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)
```

Por ejemplo, para n_steps = 3 tendríamos las siguientes variables “X” e “y”:

X -> **[[10 15]**

[20 25]

[30 35]]

Y -> **[65]**

Véase que esto tiene como longitud las cadenas de entrada, es decir, “y” llega hasta 185, siendo esto la suma de 90 y 95, últimos valores de la cadena de entrada, y “X” hace lo mismo, sigue en secuencias de longitud 3, hasta completar todos sus valores.

En cuanto al modelo, utilizamos uno muy similar al anterior:

```
#MODELO
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=2000, verbose=0)
```

La modificación que realizamos en este caso es la dimensión de la capa de entrada, que viene dada por n_input que no es otra cosa más que las dimensiones de entrada de nuestra variable.

Resultado:

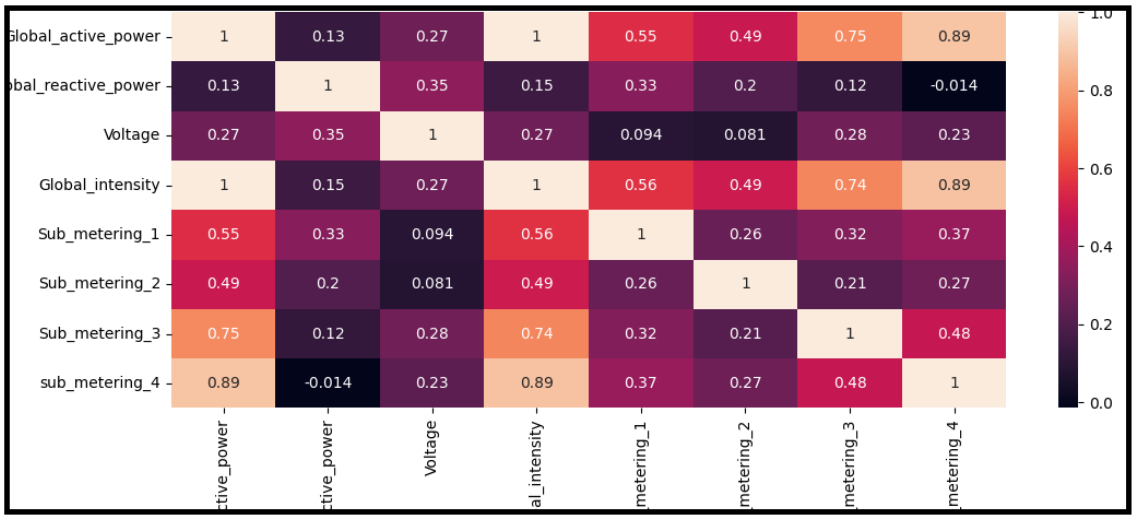
```
[[205.81075]]
```

El resultado se acerca mucho al esperado, ya que sería la suma de los dos últimos valores de entrada, en este caso [100, 105].

Nuestro dataset:

A continuación, nos disponemos a realizar la implementación con nuestro dataset para esta variante del perceptrón multicapa, el multivariable. Esto implica que, a la hora de realizar una predicción, nuestro modelo no se basará únicamente en el valor anterior de nuestra variable objetivo, sino que utilizará más de una variable para realizar este proceso.

Como ya mencionamos antes, vamos a realizar un análisis de correlación previo para partir de una base sobre la cual explicar y entender las futuras predicciones:



Mapa de calor de la correlación entre variables

Un análisis de correlación como este viene a esclarecer posibles relaciones lineales entre las distintas variables de nuestro conjunto de datos. Los valores que se observan indica que tan “fuerte” es la relación de una característica con respecto a otra, en este caso solo vamos a observar la primera columna, la relación de nuestra variable objetivo “Global_active_power” con el resto de las componentes. Cuando disponemos de una correlación positiva cercana a 1, quiere decir que cuando una variable aumente, la otra aumentará también, y cuando es -1 es lo contrario, una aumentará y la otra disminuirá o viceversa. En caso de disponer de correlaciones cercanas a 0, indica que no hay relación lineal entre el comportamiento de esas dos variables.

Antes de seguir con nuestro modelo multivariable, cabe comentar una serie de puntos para tener en cuenta en base a lo visto:

-Correlación no implica causalidad, esto quiere decir que, aunque dos variables estén muy correlacionadas como ocurre con “Global_active_power” y “Global_intensity”, esto no quiere decir que una cause a la otra.

-Relaciones no lineales, este análisis que acabamos de realizar solo tiene en cuenta las relaciones lineales entre dos variables. Al no tener en cuenta aquellas relaciones que no son de este tipo, el modelo puede subestimar la fuerza en la relación de esas dos variables.

Tras revisar este apartado, continuamos con nuestro modelo, que se queda idéntico a lo ya visto en el modelo univariante, con el cambio de los parámetros de entrada, ya que ahora aceptamos “n_features” que viene a referirse al número de características adicionales que vamos a utilizar para realizar la predicción, en este caso viene dado por el número de columnas que tiene nuestro “dataset”.

```
model = Sequential()
model.add(Flatten(input_shape=(n_steps, n_features)))
model.add(Dense(1000, activation='relu'))
model.add(Dense(700, activation='relu'))
model.add(Dense(500, activation='relu', kernel_regularizer=l2(0.01)))
model.add(Dense(1))
```

Épocas -> 100.

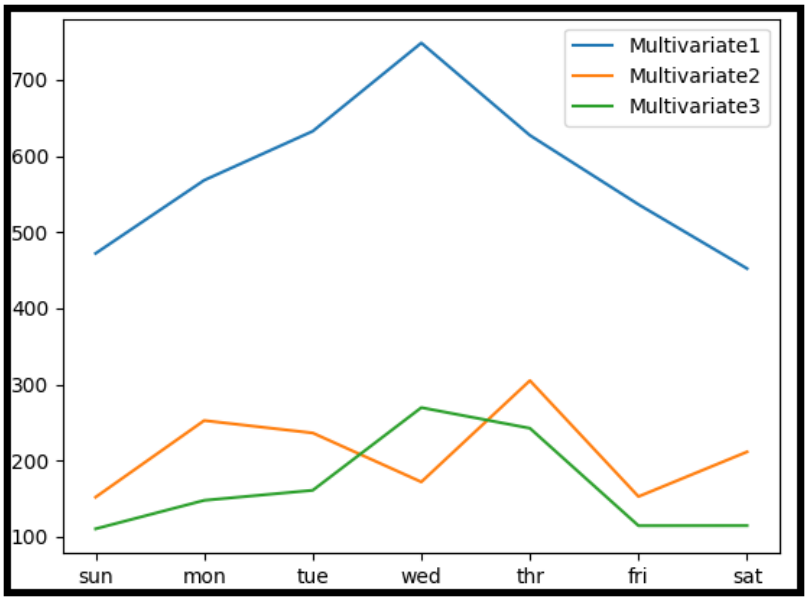
```
MLP Multivariate: [593.5] 472.2, 568.3, 632.7, 748.6, 627.3, 536.5, 452.3
```

Épocas -> 500.

```
MLP Multivariate: [177.7] 237.0, 203.2, 181.8, 202.3, 164.3, 127.8, 142.1
```

Épocas -> 1000.

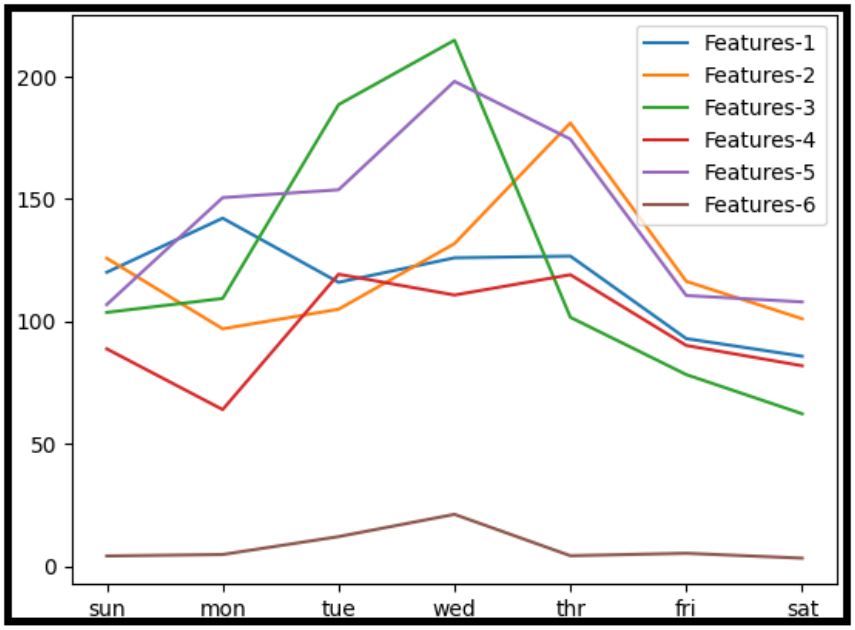
```
MLP Multivariate: [245.3] 290.1, 223.4, 320.7, 284.3, 315.1, 209.3, 236.4
```



Comparación en las distintas épocas de entrenamiento

En la comparación previa, observamos un comportamiento muy distinto al “univariate”, y es que, en primera instancia, a las 100 épocas tenemos una red que todavía no está estabilizada y por ello obtenemos resultados muy dispares, en la figura anterior solo he adjuntado uno de ellos a modo representativo. Aunque tras seguir entrenando el modelo durante más tiempo, obtenemos que la red se estabiliza y nos brinda unos resultados significantemente superiores a los vistos anteriormente.

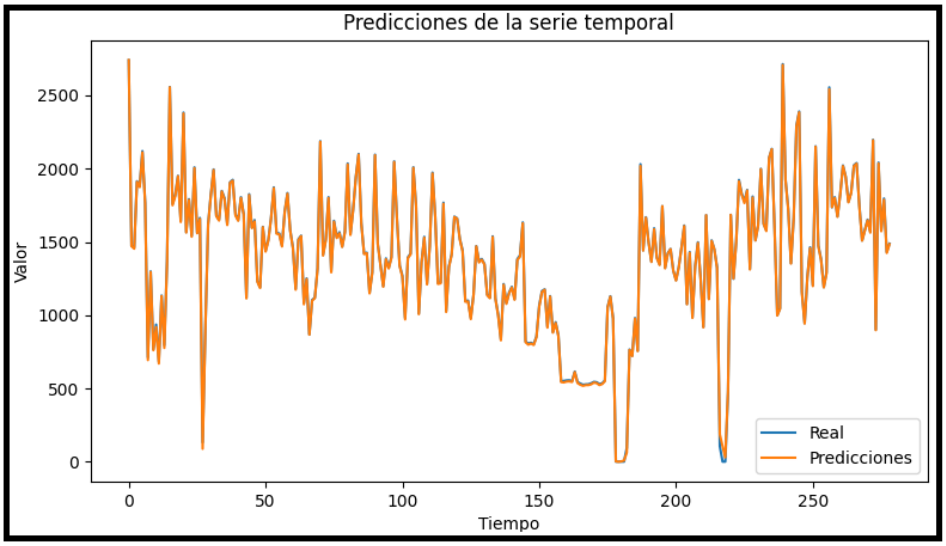
Sin embargo, antes de pasar a la comparación directa un hecho interesante a resaltar es el comportamiento cuando variamos el número de atributos que tenemos en cuenta a la hora de realizar nuestra predicción:



Comparación con distintos n-features

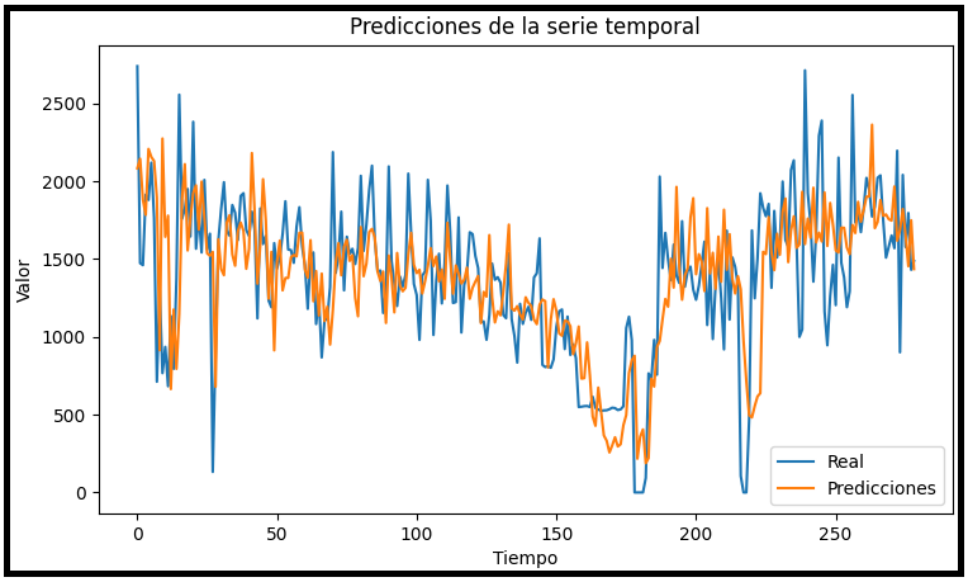
En este punto vemos un comportamiento muy interesante, y es que nuestro modelo comienza a comportarse sorprendentemente bien a medida que bajamos el número de atributos involucrados en la entrada multivariable del modelo. La disminución de variables que realizamos no es selectiva, sino que vamos recortando la última atributo en cada ejemplo. Posteriormente llevaremos a cabo una mejora de este estudio mediante una selección de variables más compleja haciendo una selección mediante algoritmos genéticos.

Si realizamos un contraste de los datos que predice la red en relación con los valores reales nos queda la siguiente gráfica para “nfeatures -6”:



Resultados reales frente a predicciones Multivariate

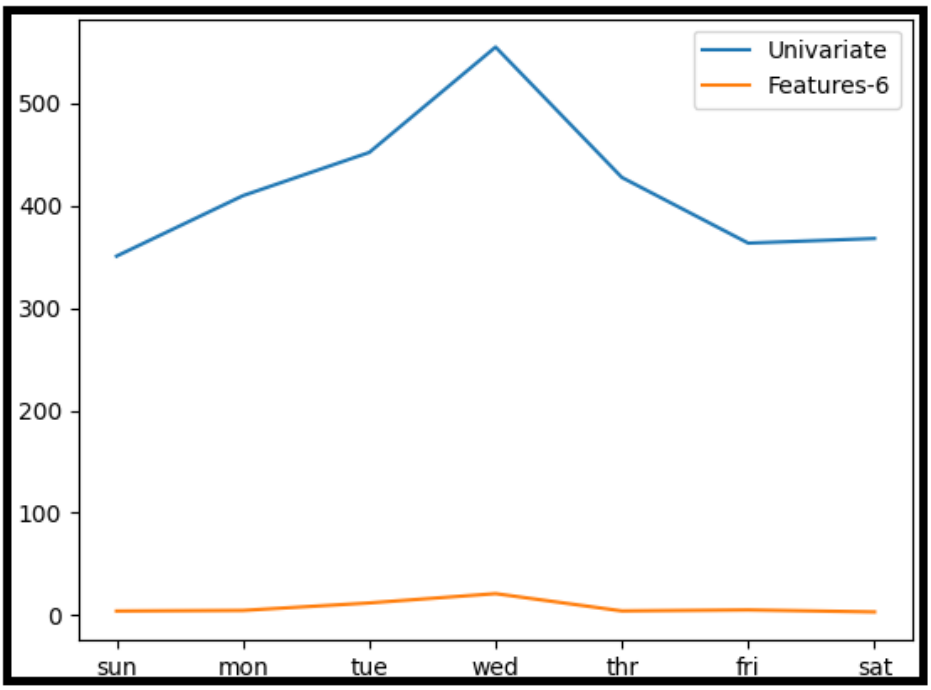
Y ahora tenemos la misma predicción para el modelo “Univariate”:



Resultados reales frente a predicciones Univariate

En esta segunda figura, podemos observar cómo tenemos una predicción que tiende a acomodarse a los resultados reales, pero que tiene su error, como veíamos en gráficas anteriores.

Por último, tenemos una comparación de los errores entre ambos:



En este apartado tenemos una clara diferencia entre el modelo “Univariate” y el actual con “features-6”, que nos brinda un error bastante reducido para el análisis de nuestra serie.

Para determinar la fiabilidad y calidad de este modelo, vamos a seguir con el estudio de sus siguientes variantes.

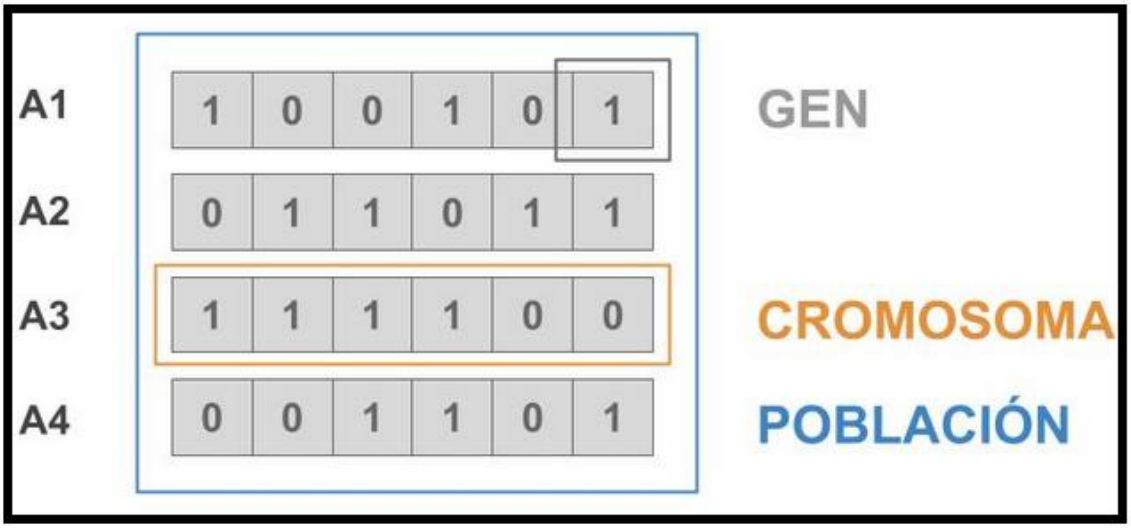
Algoritmo genético para la selección de atributos:

Nuestro punto anterior demostraba que el número de atributos utilizados para realizar el entrenamiento y pronóstico de nuestra red, tenía una fuerte influencia en los resultados en función de la selección realizada.

Partiendo de esta premisa, solo hemos realizado una selección algo trivial, quitando atributos de manera ordenada sin tener en consideración las relaciones que existen entre ellas de cara a nuestro objetivo. Para ello, vamos a realizar la implementación de un algoritmo genético para conseguir la mejor combinación de atributos que permitan maximizar nuestras predicciones.

Los algoritmos genéticos, son métodos centrados en la optimización de una función objetivo basados en la teoría de la evolución de Charles Darwin.

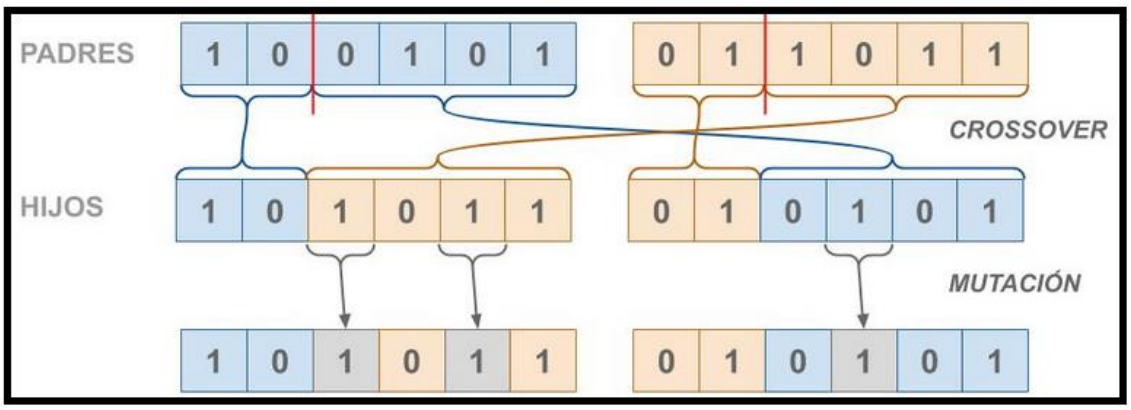
Este será nuestro punto de partida para entender esta breve introducción a los algoritmos genéticos, y es que estos se fundamentan en el intento de reproducir lo que ha ocurrido a lo largo de la historia con la evolución de “poblaciones”. Para empezar, necesitamos tener claros tres conceptos:



Algoritmo genético

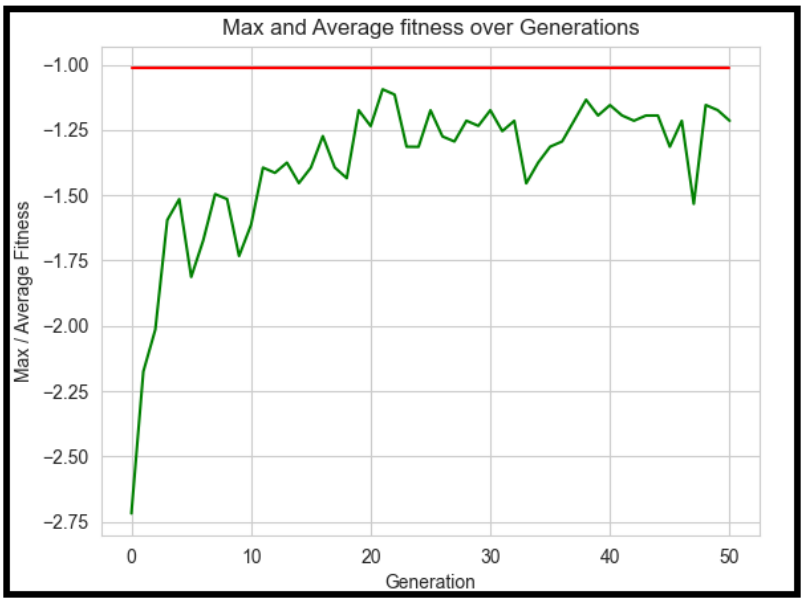
Nuestro algoritmo opera sobre una población de posibles soluciones, llamadas cromosomas, las cuales suelen ser una cadena de bits, que indican las componentes que tenemos en cuenta. Cada valor de este cromosoma, que está representado por un 1 o un 0, se le conoce como gen.

Al igual que en la teoría expuesta por Charles Darwin, la idea es generar sucesivas generaciones cada vez mejor que las anteriores. Esto se va a conseguir, seleccionando a los mejores individuos de cara población con una función fitness, la cual nos dará información sobre qué tan bueno es ese cromosoma. Posteriormente se realizarán combinaciones entre esos cromosomas para dar nuevas iteraciones que buscan mejorar a sus progenitores.



La figura anterior representa el proceso que se produce en cada iteración de nuestro algoritmo. Tenemos dos padres, que son cromosomas de la población actual, realizamos un cruce, que no es otra cosa que realizar una “mezcla” entre ambos para obtener sus respectivos hijos. Por último, tenemos una mutación, que es el proceso, por el cual en base a una probabilidad previamente asignada vamos bit por bit mutando su valor. Finalmente, estos serán los elegidos en la población actual.

Tras realizar la implementación de nuestro algoritmo genético, realizamos una ejecución con 50 épocas, y obtenemos las siguientes combinaciones de atributos para nuestro conjunto de datos:



Fitness medio y máximo durante el entrenamiento

El resultado que nos arroja el algoritmo es lo que veníamos observando cuando estábamos haciendo el análisis previo, cada vez que entrenamos con un número menos de "features"atributos, obtenemos mejores resultados. En esta ocasión esta conclusión fue encontrada con un estudio previo, no como en el apartado anterior, que fue fruto de ensayo y error en el mismo.

La siguiente etapa de nuestro estudio es utilizar redes neuronales Multi-headed.

Multi-headed:

"Multi-headed" nos indica que cada serie temporal será trabajada por redes MLP distintas, para luego ser fusionadas y así obtener una mejor salida. La siguiente gráfica muestra una representación visual de este tipo de redes:

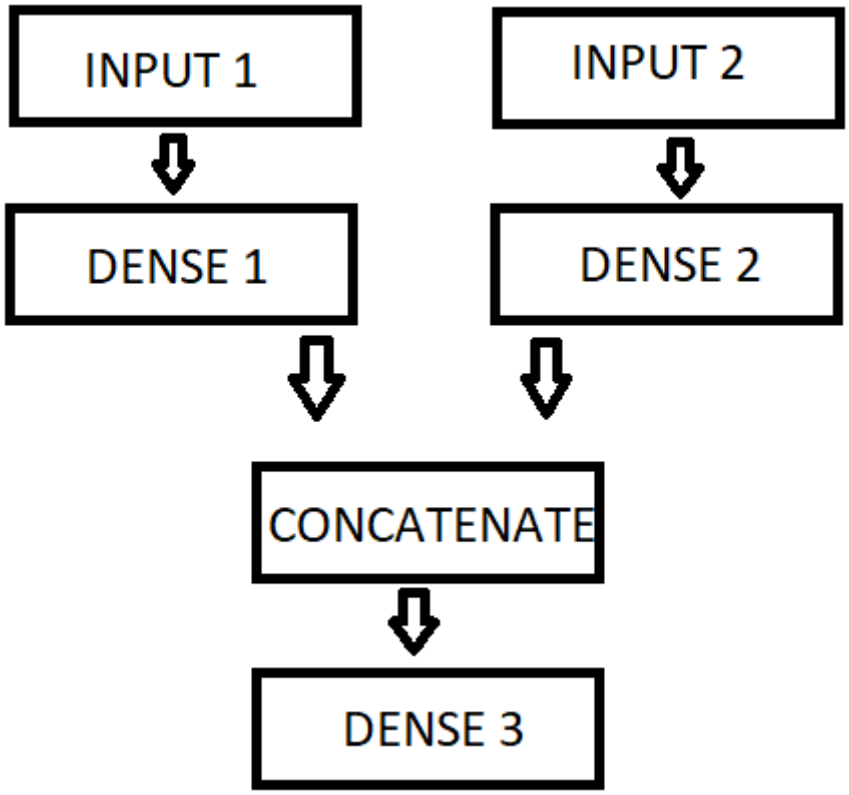


Ilustración de un modelo Multi Head

Vemos como tiene dos entradas completamente distintas, que son tratadas como los modelos univariate totalmente independientes, para luego mediante una capa unirlos y así obtener nuestra salida. Este modelo puede ser más complejo, y la figura que vemos en la parte superior omite muchos detalles de la arquitectura, pero es meramente para tener una referencia visual sobre la cual indagar en los siguientes puntos.

Implementación en Python:

En este caso, vamos a tener en esencia un problema multivariable, no obstante, las dos secuencias de entrada serán tratadas de manera separada por dos entradas distintas. Por ejemplo:

-secuencia 1-> [10, 20, 30]

-secuencia 2-> [15, 25, 35]

Estas dos secuencias, se tratarán de la misma manera que en la variante multivariable. Generamos la secuencia de salida, que viene dada por la suma de los dos elementos en cada posición del array, posteriormente ajustamos las dimensiones para luego apilarlas con el ya conocido método de numpy “hstack()”. Definimos nuestra longitud de ventana “n_steps” para enviarla al método “Split_squences()” y así obtener las subsecuencias de longitud “n_steps” y la salida esperada de las mismas.

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])

in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))

dataset = hstack((in_seq1, in_seq2, out_seq))
n_steps = 3
X, y = split_sequences(dataset, n_steps)
```

Tras tener nuestra entrada y salida de la red, vamos a separar la entrada en dos subconjuntos, ya que, en este caso, nuestra red Multi-headed tendrá dos entradas.

```
X1 = X[:, :, 0]
X2 = X[:, :, 1]
```

Tras tener este apartado realizado, procedemos a crear el modelo de nuestra red. No dista mucho de lo visto anteriormente:

```
visible1 = Input(shape=(n_steps,))
dense1 = Dense(100, activation='relu')(visible1)

visible2 = Input(shape=(n_steps,))
dense2 = Dense(100, activation='relu')(visible2)

merge = concatenate([dense1, dense2])
output = Dense(1)(merge)
model = Model(inputs=[visible1, visible2], outputs=output)

model.compile(optimizer='adam', loss='mse')
model.fit([X1, X2], y, epochs=2000, verbose=0)
```

En primer lugar, creamos una capa de entrada “visible1”, con una longitud de “n_steps”, posteriormente creamos una capa Dense totalmente conectada con la anterior, con 100 neuronas y función de activación “relu”. Hacemos esto mismo para la otra entra de la red, visible2. Luego concatenamos las salidas de las dos capas densas anteriores utilizando la función merge de Keras. Por último, añadimos una capa de salida con una única neurona, ya que se trata de un modelo “single-step”..

Por último, compilamos el modelo y lo entrenamos con los valores ya explicados.

Como resultado obtenemos lo siguiente de utilizar los siguientes valores como entrada:

-entrada -> [[80, 85], [90, 95], [100, 105]]

La salida como hemos explicado debería ser la suma de los dos últimos valores de esta sucesión, es decir, 100 y 105.

```
[[205.76218]]
```

Nuestro valor resultado se acerca bastante a lo esperado.

Resultado con nuestro conjunto de datos:

Para el análisis de nuestro conjunto de datos con esta variable del perceptrón multicapa, al igual que con el resto, hemos intentado seguir el mismo esquema realizado anteriormente, conservar lo más similar el modelo en cuanto a la distribución de capas y configuración de estas, para ver el comportamiento de nuestro dataset de la manera más arbitraria posible.

Recordemos que este "Multi-Headed" se fundamenta en tener varias entradas, las cuales luego de juntan en una capa "merge" para dar una única salida en esta red neuronal. El código del modelo en Python sería el siguiente:

```
input_layer = Input(shape=(n_steps, n_features))
input_layer_flatten = (Flatten())(input_layer)
Dense1 = Dense(500, activation='relu')(input_layer_flatten)
Dense2 = Dense(500, activation='relu')(input_layer_flatten)
merge = concatenate([Dense1, Dense2])
Dense3 = Dense(700, activation='relu')(merge)
Dense4 = Dense(500, activation='relu', kernel_regularizer=l2(0.01))(Dense3)
DenseF = Dense(1)(Dense4)
```

Épocas -> 100.

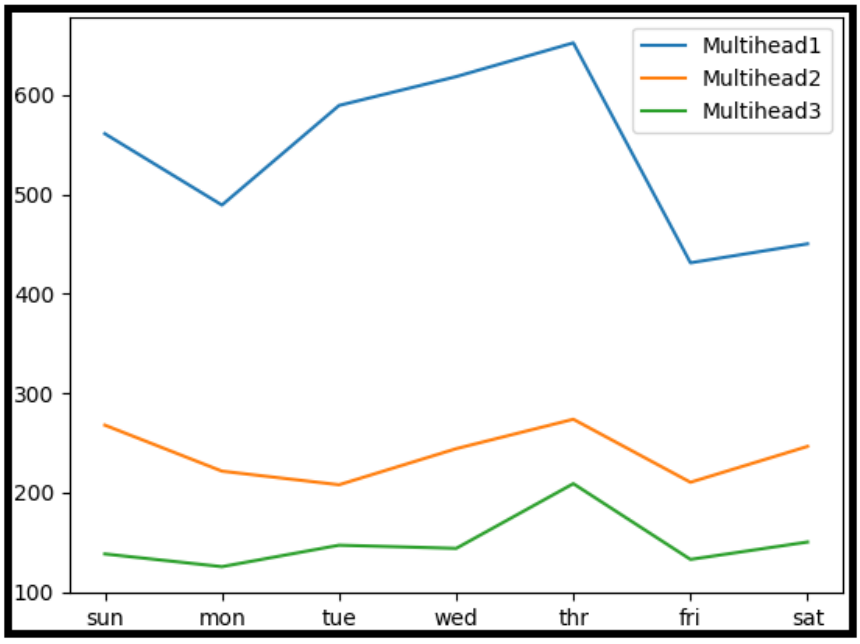
```
MLP MultiHead: [524.7] 561.1, 489.3, 589.5, 618.5, 652.5, 431.2, 450.3
```

Épocas-> 500.

```
MLP MultiHead: [199.7] 244.4, 305.9, 209.8, 229.1, 203.1, 173.2, 140.7,
```

Épocas->1000.

```
MLP MultiHead: [149.1] 138.3, 125.5, 147.1, 143.9, 209.0, 132.8, 150.3
```



En este caso vemos comportamientos similares al Multivariate en cuanto a los resultados de los errores, y debido a la similitud de esta variante con la anterior mencionada, ya que ambas utilizan “n-features” para realizar sus predicciones, vamos a realizar un análisis adicional restando el número de “features” y viendo los resultados:

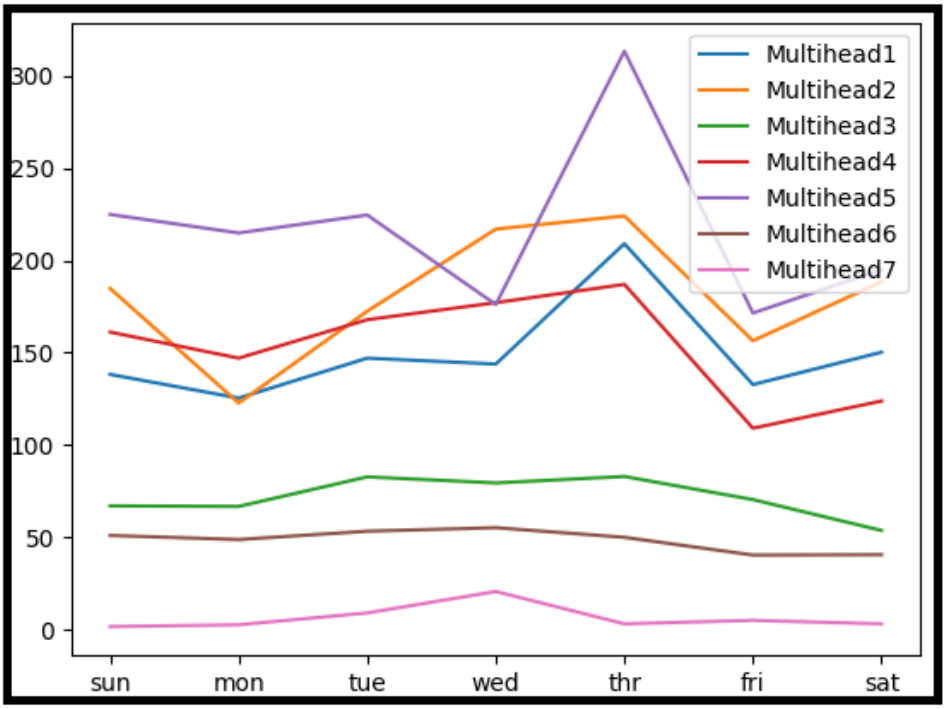
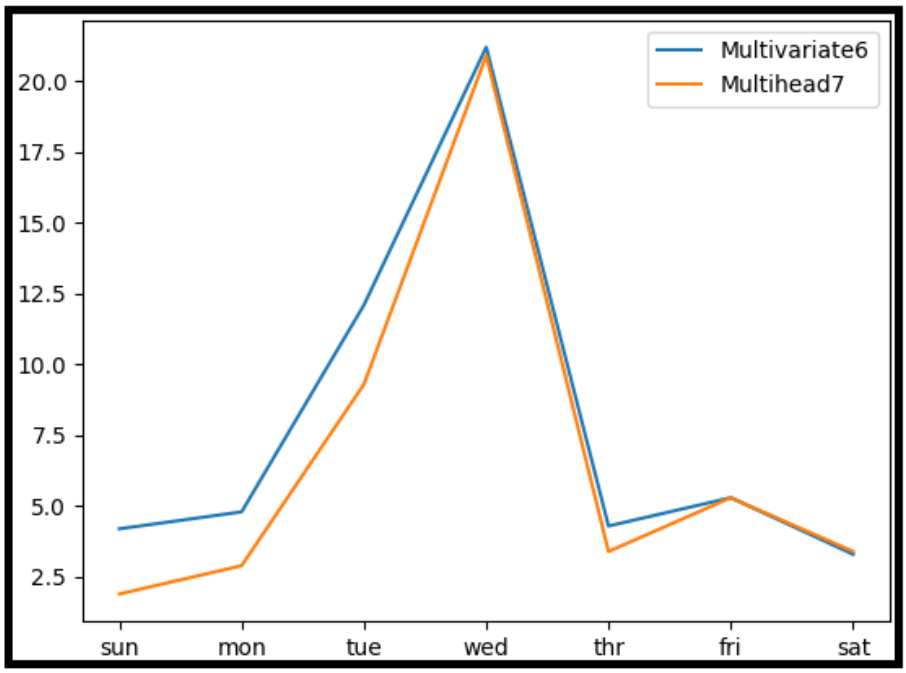


Ilustración comparativa cambiando el “n-features”

Tenemos unos resultados similares a los estudiados en el multivariable, al bajar el número de features utilizadas tenemos unos mejores resultados. Los que tenemos con “n_features-7” son casi idénticos a los que daban en el mutlivariable, como se observa en la siguiente figura:



Multi Variate frente al Multi Head

Esta última gráfica la vamos a tomar como la comparación directa con el anterior modelo.. En este caso volvemos a encontrar un panorama similar donde no conseguimos distanciarnos de la variante anterior del MLP.

Multiple Parallel Series

Este modelo, variante del MLP, tiene como característica que involucra múltiples variables. La diferencia con lo estudiado anteriormente radica en que el modelo indica una estructura paralela en la que cada variable dispone de su propia "red", para posteriormente combinar sus salidas y realizar una predicción conjunta. Cada red de las mencionadas dispone de independencia con respecto a las otras, lo que nos da libertad en cuanto a parámetros y a estructura. Este enfoque permite a las redes, compartir información y aprender entre ellas para así poder tener en cuenta relaciones más complejas entre las variables.

Este modelo es especialmente útil cuando se trabaja con conjuntos de datos que dispongan de varias variables y relaciones complejas entre ellas. Ya que la virtud de este radica en la capacidad de procesarlas por separado y a su vez conseguir relaciones intrínsecas entre las mismas.

Implementación en Python:

Para este caso, volvemos a tener un esquema similar al multivariante, dos secuencias de entrada, que cuando sumas sus elementos nos da una de salida.

-secuencia 1-> [10, 20, 30]

-secuencia 2-> [15, 25, 35]

-salida -> [25, 45, 65]

Para este caso, queremos que nuestra predicción nos devuelva un resultado del tipo:

-predicción-> [40 45 85]

Siendo esta la continuación aparente de nuestra serie temporal.

Para realizar este proceso, es necesario realizar pequeñas modificaciones en nuestros métodos:

En primer lugar, definimos nuestras series, para a continuación generar la secuencia de salida. Posteriormente, ajustamos las dimensiones para que sean aptas para la entrada de nuestra red y las juntamos utilizando el método “hstack()” de numpy.

Por último, fijamos el valor de “n_steps” para realizar las subsecuencias en “split_squences()” y finalmente realizamos el cálculo de la dimensión de entrada de la red y reajustamos nuevamente los datos de entrada definidos en la variable “X”, tal y como hemos venido haciendo.

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))

dataset = hstack((in_seq1, in_seq2, out_seq))

n_steps = 3
X, y = split_sequences(dataset, n_steps)
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
```

A la hora de declarar el modelo, mantenemos todos los parámetros intactos con respecto a los ejemplos anteriores y únicamente modificamos la dimensión de la capa densa destinada a la salida de la red, como si de una “Multi-step” se tratara. En este caso, ponemos como dimensión de salida, la de la secuencia de salida obtenida en el método “split_sequences()”, que tiene valor 3.

```
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(y.shape[1]))
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=2000, verbose=0)
```

En base a esto, obtenemos un total de tres parámetros de salida, como ya veníamos anticipando:

```
[[100.067505 105.30532  205.42313 ]]
```

Los valores utilizados para realizar esta predicción son -> [[70,75,145], [80,85,165], [90,95,185]]. Con lo que podemos determinar que la predicción está realmente cerca de los valores esperados.

Nuestro dataset:

Utilizando el modelo explicado en el apartado anterior para la predicción de los valores del array, vamos a implementar ese modelo en el código base que estamos utilizando en el resto de modelo y así poder tener el mismo formato de valores resultados para poder compararlos entre sí:

Épocas 100 ->

```
MLP Parallel Series:: [427.5] 360.7, 350.3, 370.1, 344.0, 447.5, 369.8, 388.9
```

Épocas 500->

```
MLP Parallel Series:: [506.1] 442.8, 498.3, 434.4, 499.2, 467.2, 433.2, 408.3
```

Épocas 1000->

```
MLP Parallel Series:: [595.9] 577.2, 510.0, 526.5, 549.3, 557.5, 472.8, 545.7
```

Vemos resultados similares en comparación al resto de modelos, en 100 épocas de entrenamiento es donde obtenemos los mejores resultados del modelo, ya que para esta red tan “simple” es suficiente para entrenarla y que no se produzca un sobre entrenamiento como vemos en las siguientes.

Si miramos la comparación directa con los resultados anteriores vemos comportamientos similares:

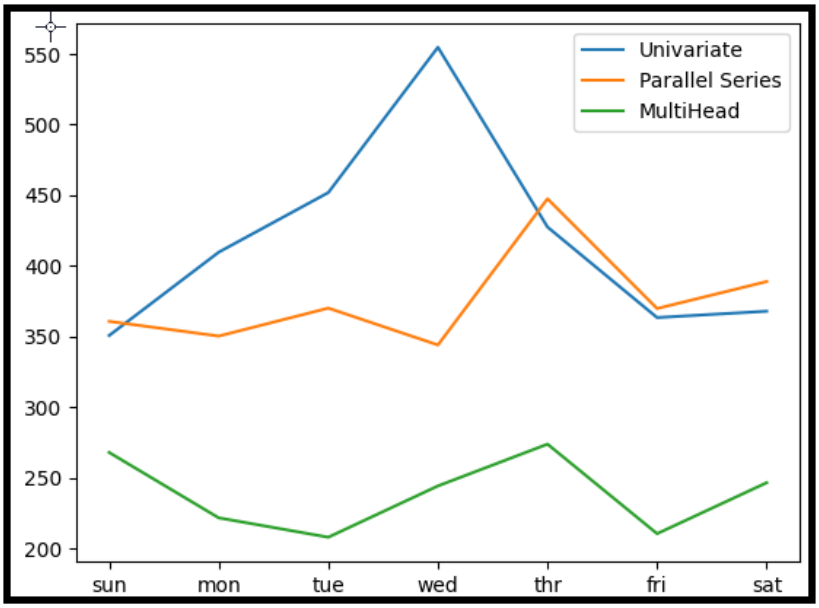


Ilustración comparativa entre los distintos modelos

Este modelo se queda con unos valores similares al Univariate, pero no consigue rendir al nivel que nos presenta el modelo Multi-headed.

Multi-output:

Similar a lo comentado anteriormente, este modelo se fundamenta directamente en la idea de disponer de varias salidas al mismo tiempo. La parte interesante que nos plantea este acercamiento es la posibilidad de que esas salidas estén relacionadas o no. Cada capa de salida en este modelo se entrena para predecir una variable de salida diferente, cada capa puede tener su propia arquitectura de red neuronal, lo que permite completa flexibilidad para este punto.

En nuestro caso, no se llega a explotar estas ventajas ya que utilizaremos la misma arquitectura y los datos están muy relacionados entre sí, como hemos venido observando.

Implementación en Python:

Para la implementación, volvemos a partir del mismo punto que el modelo anterior. Dos secuencias de entrada, que sumadas dan una de salida:

-secuencia 1-> [10, 20, 30]

-secuencia 2-> [15, 25, 35]

-salida -> [25, 45, 65]

No obstante, ahora cada parámetro de los tres de salida será dado por una salida distinta de la red. Vamos a disponer de tres salidas en este caso.

En cuanto al código en Python, volvemos a tener exactamente el mismo planteamiento. Creamos las tres secuencias, las dos de entrada y la consecuente de sumar las dos anteriores, reajustamos las dimensiones para generar como arrays de 1 columna para así apilarlos posteriormente en la variable dataset. Luego llamamos como siempre al método “split_sequences()”, para realizar la subdivisión y así disponer de una mayor cantidad de datos para entrenar nuestra red.

Por último, llega la parte explícita de este modelo “Multi-output”, la cual es necesaria ya que esta variante necesita tres arrays de salida distintos por muestra. En otras palabras, lo que hacemos es dividir nuestra variable de salida “y” en tres variables separadas “y1”, “y2” e “y3” que corresponden a cada columna de la matriz “y”.

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))

dataset = hstack((in_seq1, in_seq2, out_seq))

n_steps = 3
X, y = split_sequences(dataset, n_steps)
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))

y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
```

En cuanto al modelo, vemos similitudes con el modelo “multi-headed”, pero ahora en sentido “inverso”, ya que es una entrada que se separa en varias salidas. En las siguientes líneas vemos como creamos una capa de entrada, con la dimensión “n_input”, como veníamos haciendo en todos los ejemplos. Posteriormente, se conecta con una capa densa de 100 neuronas, que se divide en tres salidas.

Compilamos el modelo, indicando que el output viene dado por tres capas distintas y lo entrenamos.

```
visible = Input(shape=(n_input,))
dense = Dense(100, activation='relu')(visible)
output1 = Dense(1)(dense)
output2 = Dense(1)(dense)
output3 = Dense(1)(dense)
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
```

Por último, vemos los resultados para los mismos valores que el anterior ejemplo, aunque en este caso son arrojados por tres salidas independientes:

```
[array([[99.57944]], dtype=float32), array([[105.68435]], dtype=float32), array([[206.50648]], dtype=float32)]
```

Los valores utilizados para realizar esta predicción son -> [[70,75,145], [80,85,165], [90,95,185]]. Vemos que son muy similares a los estudiados en el modelo anterior.

Multi-step:

La siguiente variación de nuestro perceptrón viene dada por la variable de salida de la red, anteriormente, ya habíamos probado con una o varias variables de entrada, lo cual, nos desembocó en dos modelos distintos. En el caso que nos acontece ahora vamos a tener como salida una secuencia de valores en lugar de una única predicción. En otras palabras, vamos a predecir más de un instante en el tiempo cuando hagamos nuestro pronóstico.

Implementación en Python:

Como hemos comentado en los casos anteriores, primero vamos a ver el formato de los datos que vamos a emplear y lo que esperamos obtener, para así poder comentar posteriormente las modificaciones que ha sufrido nuestro código con relación a esto.

En este caso volvemos a tener una única variable de entrada:

-secuencia -> [10 20 30]

Y esperamos una variable de salida, pero con longitud definida por una nueva variable llamada “n_steps_out”, que como podemos intuir cumple una función similar a la ya conocida “n_steps”, que en este caso pasa a llamarse “n_steps_in” para así evitar confusiones. Para el siguiente ejemplo tomemos como la secuencia de entrada, la mencionada y como salida utilizaremos “n_steps_out = 2”, con lo que quedaría:

-salida -> [40 50]

En cuanto al código en Python es muy similar al visto en el modelo “Univariate”, no obstante, en la parte de dividir el conjunto de datos y dar las dimensiones a la capa de salida cambia.

```
def split_sequences(sequences, n_steps_in, n_steps_out):
    X, y = list(), list()
    for i in range(len(sequences)):
        end_ix = i + n_steps_in
        out_end_ix = end_ix + n_steps_out
        if out_end_ix > len(sequences):
            break
        seq_x, seq_y = sequences[i:end_ix], sequences[end_ix:out_end_ix]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)
```

El método ahora recibe como parámetro los ya mencionados “n_steps_in” y “n_steps_out” encargados de crear las subsecuencias de entrada “in” y salida “out”. Modificamos el bucle para que termine al completar la secuencia de salida, ya que en este caso no es un único valor.

Las variables “X” e “y” quedaría de la siguiente manera:

-X -> [[10 20 30]

[20 30 40]

….

-y-> [[40 50]

[50 60]

….

Observamos que ahora tenemos una predicción de dos valores en lugar de uno.

En cuanto al modelo, volvemos a casi repetir lo visto anteriormente, modificando como se puede intuir, la salida de nuestra red:

```
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps_in))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=2000, verbose=0)
```

Todo se ajusta a lo visto en variantes anteriores del MLP, pero ahora en la capa “Dense” de salida en lugar de disponer de un “1” indicando que solo retorna un valor de salida, tenemos “n_steps_out”, dando dos valores en este caso para la salida.

Resultados para la entrada [70, 80, 90]:

```
[[103.14431 115.22747]]
```

Vemos como nos devuelve dos valores, intentando seguir la secuencia de aumentar en 10 cada valor.

Nuestro dataset:

Para este punto, repetimos lo ya antes visto. La única diferencia en el modelo es el parámetro que le pasamos a la última capa densa de salida del modelo, ya que hasta ahora esta era “1”, indicando que simplemente nos daba una predicción para el siguiente instante de tiempo, es decir, para el siguiente “step”.

Con lo cual ahora tenemos dos parámetros “n_steps”, el ya conocido que se encarga de ajustar la ventana de tiempo de los datos de entrada, y el nuevo “n_steps_out”, que como propio nombre indica, va ligado a la función de conseguir que nuestro modelo devuelva “n” valores en el futuro a predecir.

```python
def keras_forecast_multistep(train, test, n_steps,n_steps_out):
    train_x, test_x = split_dataset(train)
    train_y, test_y = split_dataset(test)
    model = Sequential()
    model.add(Dense(1000, activation='relu',input_dim=n_steps))
    model.add(Dense(700, activation='relu'))
    model.add(Dense(500, activation='relu', kernel_regularizer=l2(0.01)))
    model.add(Dense(n_steps_out))

    model.compile(optimizer='adam', loss='mse')

    model.fit(train_x, train_y, epochs=500, verbose=1)
```

Épocas -> 100.

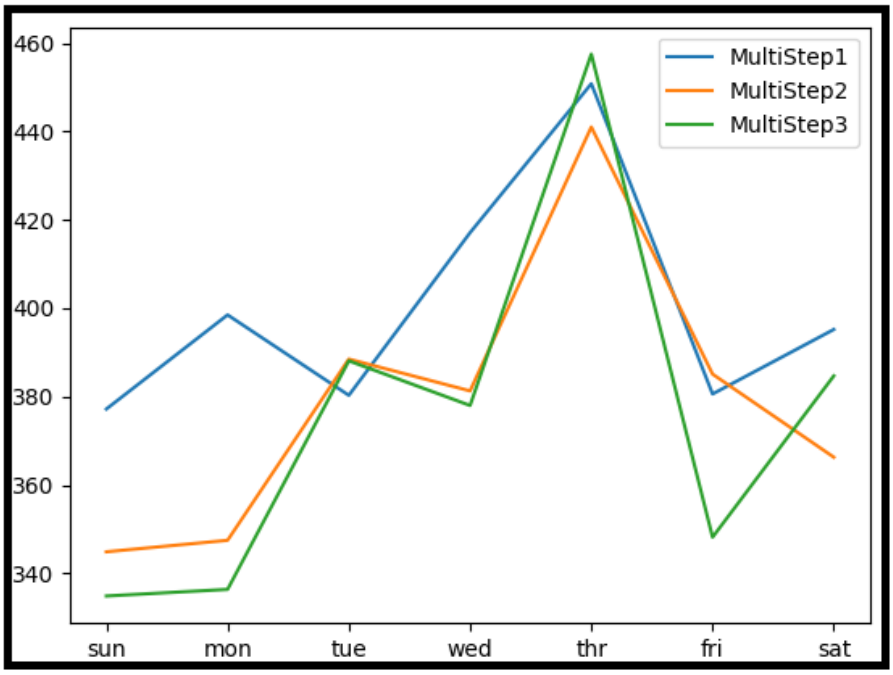
```
MLP MultiStep: [452.5] 377.2, 398.5, 380.3, 417.1, 450.8, 380.6, 395.2
```

Épocas -> 500.

```
MLP MultiStep: [435.9] 344.9, 347.5, 388.5, 381.3, 441.0, 385.1, 366.3
```

Épocas -> 1000.

```
MLP MultiStep: [430.9] 334.9, 336.4, 388.1, 378.0, 457.5, 348.2, 384.7
```



Para este caso, vemos que independientemente de los valores que indiquemos en el número de épocas a entrar, los resultados se quedan bastante similares. Para profundizar en este comportamiento dado por el modelo “MultiStep”, hemos decidido realizar entrenamientos más allá de las mil épocas, para ver el comportamiento que

surten en el modelo, ya que recordemos, en los anteriores modelos se observaba una pérdida apreciable en el rendimiento llegadas a las mil épocas de entrenamiento.

Adjuntamos el resultado de entrenar el modelo durante mil épocas:

```
MLP MultiStep: [423.4] 349.7, 347.8, 377.6, 369.9, 457.9, 357.1, 363.1
```

Como se ve en claro, el resultado es idéntico a lo observado anteriormente. Llegados a este punto, es necesario revisar la estructura de los datos y distribución que le estamos dando a este modelo, no obstante, esto se realizará en el apartado final de este capítulo donde se realizarán comparaciones más en detalle de todos los modelos y los vistos en el capítulo anterior.

Aunque los resultados obtenidos por el modelo en sus distintos entrenamientos sean similares, no quiere decir que no sean válidos para el problema que nos acontece, ya que podemos ver en la siguiente comparación que rinde mejor que su predecesor "Univariate" a la hora de realizar el pronóstico de nuestro conjunto de datos. Aunque cuando lo medimos con nuestro caballo ganador, el modelo "multihead", vemos que vuelve a quedarse un escalón por debajo de este.

Multivariate Multi-step:

Para finalizar esta parte de MLP, vamos a realizar una combinación de dos variantes ya mencionadas y estudiadas en los apartados anteriores, la "Multivariate" y "Multi-step".

Implementación en Python:

Como recordamos, la implicación que tienen las características mencionadas es que nuestra red tendrá como entrada más de una única variable, véase 2 secuencias y la salida no será un valor único. Para ilustrar esto vamos a poner un ejemplo como en los casos anteriores:

-secuencia 1-> [10, 20, 30]

-secuencia 2-> [15, 25, 35]

Estas dos secuencias serían la entrada de la red, y como salida esperamos la suma de los dos últimos valores de las secuencias y la suma de los dos futuros valores de la serie:

-salida -> [65, 85]

Que es el resultado de sumar 30 + 35 y 40 +45, que en este caso serían los siguientes valores de la serie temporal.

En cuanto al código utilizado, es una unificación de lo ya antes visto. Repasemos brevemente el mismo:

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))

dataset = hstack((in_seq1, in_seq2, out_seq))

n_steps_in = 3
n_steps_out = 2
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
```

En primer lugar, volvemos a tener dos secuencias de entrada, al tratarse de un modelo multivariable. Posteriormente, generamos la secuencia de salida, que viene dada por la suma de los elementos de las cadenas de entradas en la posición relativa de cada uno. Los transformamos en matrices de una columna para poder apilarlos y así tenerlos en el formato deseado de entrada en la red.

Seguidamente, tenemos nuestras variables de “n_steps” tanto de entrada como salida, estas son enviadas al método “split_sequences()” que se encarga de darnos la entrada y salida esperada de la red.

```
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
model.fit(X, y, epochs=2000, verbose=0)
```

Por último, el modelo es igual a los anteriores, tomando como entrada la dimensión de los datos y como salida “n_steps_out”. Por último, tras entrenar la red obtenemos lo siguiente:

```
[[186.26408 207.3914 ]]
```

Esto es para la entrada -> [[70, 75], [80, 85], [90, 95]]. Los valores tendrían que ser 90 + 95 = 105 y 100 + 105 = 205, vemos que nuestra red se acerca bastante a los valores esperados.

Nuestro dataset:

Tras haber visto el modelo “Multivariate”, el cual nos brindaba la característica de poder tener más de una variable de entrada a la hora de predecir. Y por otro lado, el modelo “Multi-Step” que nos daba la capacidad de no limitarnos solo a predecir el siguiente paso en el tiempo de nuestra serie temporal, sino que abría el abanico a poder predecir “n” valores futuros, obtenemos la combinación de ambos que resulta en este modelo “Multivariate Multi-step”.

Se observan similitudes obvias con lo mencionado anteriormente, las cuales pasan por el “n_steps_out” para determinar el número de elementos a predecir en el futuro y la capa de entrada con “n_features”, que indica que no utilizamos simplemente la variable a predecir como dato para entrenar.

```
train_x, test_x = split_dataset(train)
train_y, test_y = split_dataset(test)
model = Sequential()
model.add(Flatten(input_shape=(n_steps, n_features)))
model.add(Dense(1000, activation='relu'))
model.add(Dense(700, activation='relu'))
model.add(Dense(500, activation='relu', kernel_regularizer=l2(0.01)))
model.add(Dense(n_steps_out))

model.compile(optimizer='adam', loss='mse')

model.fit(train_x, train_y, epochs=100, verbose=1)
```

Épocas -> 100.

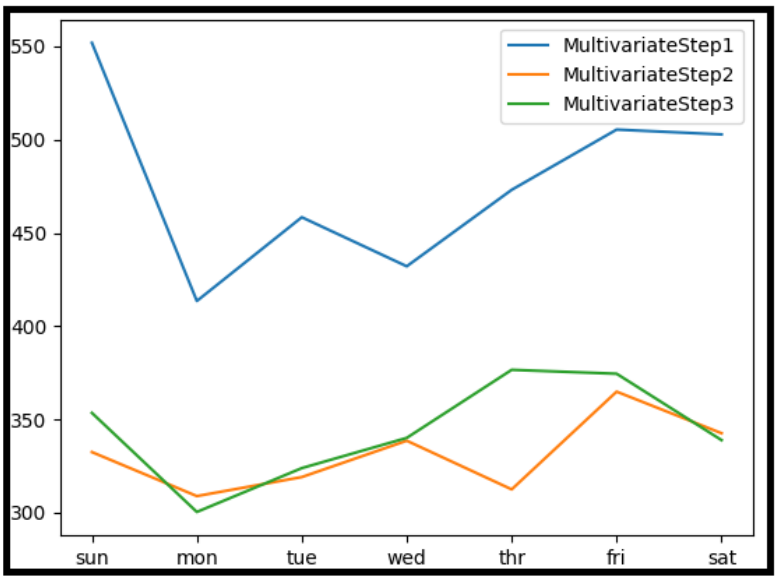
```
MLP MultiStepM: [564.4] 551.9, 413.5, 458.4, 432.1, 473.1, 505.4, 502.8
```

Épocas -> 500.

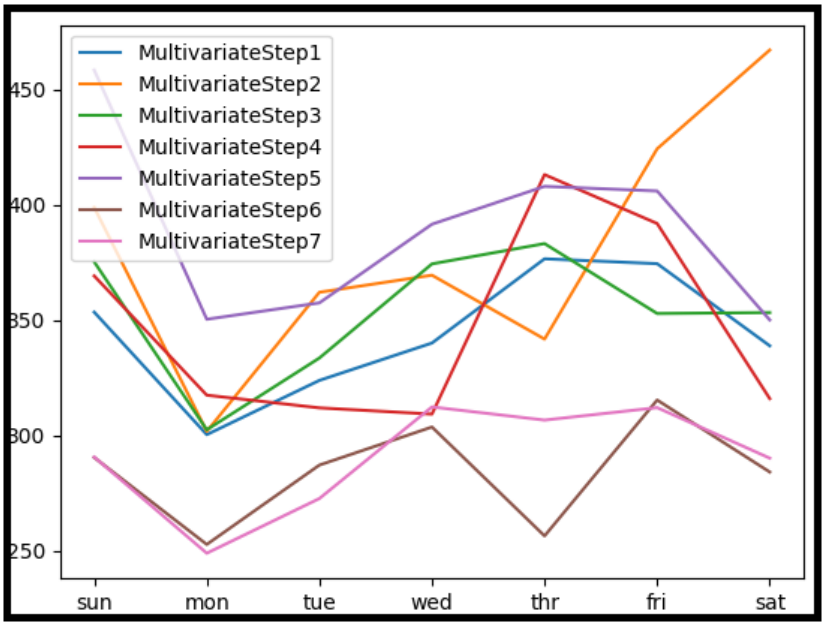
```
MLP MultiStepM: [371.6] 332.0, 308.6, 319.3, 338.4, 312.3, 364.4, 342.6
```

Épocas -> 1000.

```
MLP MultiStepM: [385.2] 353.5, 300.4, 323.9, 340.1, 376.6, 374.5, 338.9
```



Podemos volver a apreciar resultados que entran en la media que están dando los modelos en líneas generales, sin embargo, para tratar este tipo de modelos como se trata de uno "multivariate" vamos a volver a analizar el comportamiento variando el número de "features":



El comportamiento vuelve a ser el ya estudiado, a menor número de componentes mejor rendimiento obtenemos. Aunque estos resultados no son tan imponentes como los vistos en la variable multihead, por ejemplo, cómo podemos comprobar a continuación:

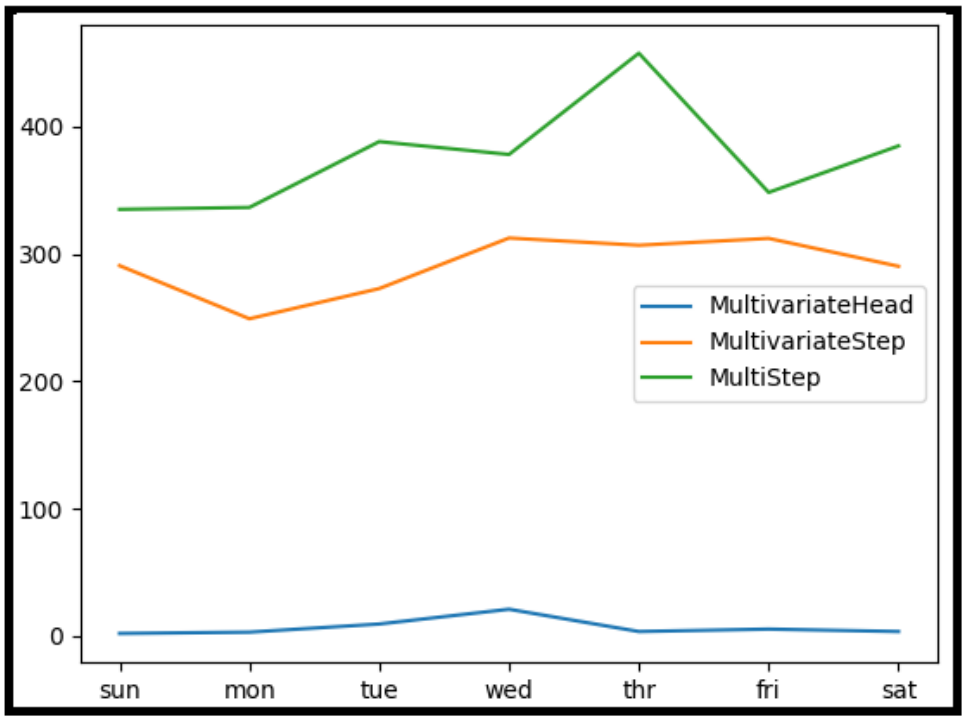


Ilustración comparativa entre los distintos modelos

Hemos añadido el modelo "multistep" clásico, el que utiliza únicamente una variable de entrada a la gráfica para tener un punto de referencia de la situación en la que se encuentra este modelo "multivariate-multistep", el cual se queda a medio camino entre ambos.

Con estos resultados podemos esclarecer que esta variante se vuelve a beneficiar de utilizar más características del "dataste" para mejorar sus resultados en comparación a su modelo "univariable". Pero que los modelos "multistep" no llegan a bridar los resultados tan cercanos a los reales, como puede ser en el modelo multihead.

CNN Y LSTM:

En el ámbito del análisis de series temporales, un punto vital y crucial para poder realizar pronósticos acertados es conseguir capturas y observar patrones en las mismas, para este punto las Redes Neuronales Convolucionales (CNN) y las Memorias a Largo Plazo de Redes Neuronales (LSTM) han demostrado desenvolverse con cierta solvencia en este punto. Echando la vista atrás con nuestro Perceptrón Multicapa, estas nuevas arquitecturas nos brindan características únicas que nos ayudan a abordar de mejor manera nuestra predicción de series temporales.

La principal diferencia donde van a brillar estas nuevas arquitecturas es en una cualidad que no posee el MLP, no son buenas para capturar información a largo plazo. Esto para el análisis de series temporales es crucial, la capacidad de captar patrones o dependencias a lo largo de la serie temporal.

LSTM:

La primera parada de este apartado serán las LSTM o Memorias a Largo Plazo de Redes Neuronales, estas son arquitecturas especializadas en el procesamiento de series temporales. Si las comparamos con las redes neuronales tradicionales, las LSTM están diseñadas para captar dependencias a largo plazo en un conjunto de datos secuenciales, como comentamos en el punto anterior. La clave para lograr este hito son las células de memoria que posee y que permiten almacenar la información relevante a lo largo de la secuencia. Como venimos estudiando esto es una cualidad ideal para predecir dependencias futuras en base a dichos patrones.

Vamos a estudiar el elemento que hace posible este modelo, las células de memoria, que tendrían la siguiente disposición:

-Input gate(entrada): Como su nombre indica es la entrada que controla que información nueva tenemos que guardar en nuestra célula de memoria. Actúa a modo de filtro que decide qué información es relevante y debe ser guardada en nuestra célula. El cálculo se obtiene a partir de la entrada actual y la información contextual anterior.

-Forget gate(olvido): En este caso determina que información almacenada en la célula de memoria debe ser eliminada. El objetivo de esta "forget gate" es ayudar a mantener solo la información relevante dentro de la célula de memoria y descartar la información obsoleta. Al igual que en la "input gate" se calcula en base a la entrada actual y la información contextual anterior.

-Output gate(salida): Determina que información disponible en nuestra célula debe de ser utilizada para la salida de nuestra red neuronal. Su función es controlar en qué medida la información almacenada será influyente o no para la salida de la LSTM. En este caso se calcula con la entrada actual y la información contextual previa.

La célula de memoria se fundamenta en la combinación de estas tres partes, la cual permite a nuestra red LSTM tener información relevante a lo largo de secuencia y actualizar de manera adecuada.

Para comprender mejor el comportamiento de las LSTM, en primer lugar, tenemos que entender cómo va a ser el flujo de una arquitectura RNN, y ver cómo influye las células LSTM en la misma:

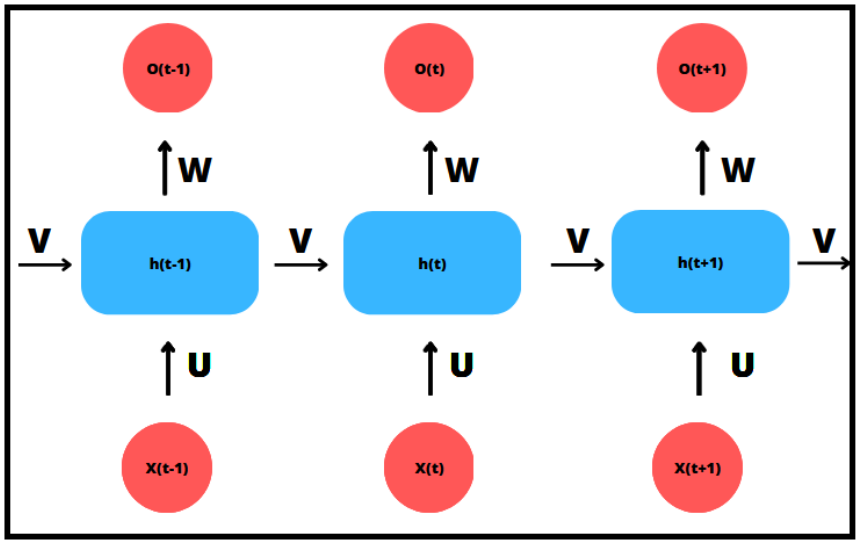
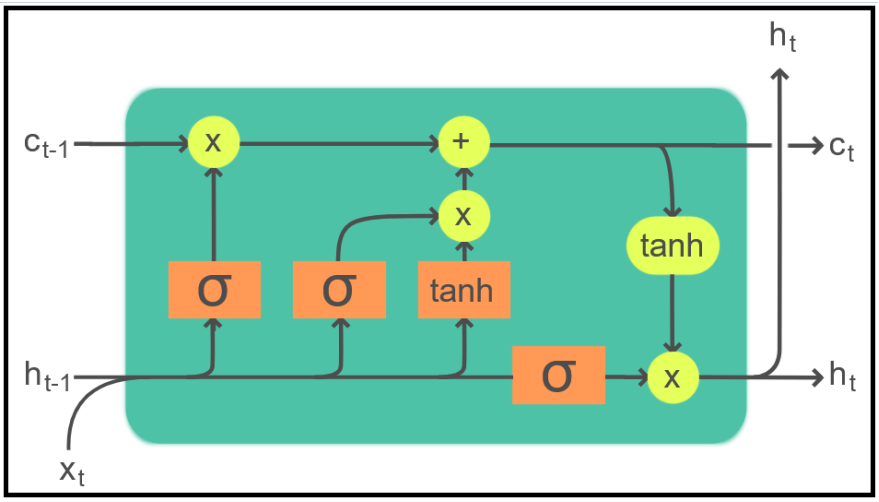


Diagrama de flujo

En el siguiente diagrama vemos como tenemos diferentes estados, y como se ven afectados por el estado anterior y la entrada actual que tenga el mismo. Esto es importante y cobra relevancia debido al problema al que se enfrentan las RNN cuando tenemos secuencias extremadamente largas, y es que cuando disponemos de un valor "X", que sea inferior a 1, por ejemplo, 0.98 a la larga tenderá a valer 0, esto es lo que se conoce como desvanecimiento del gradiente. Cuando ocurre el caso opuesto, y el valor es superior a 1, este tiende a infinito. Para nuestras series temporales esto puede ser un problema, es por ello por lo que las células de memoria en las LSTM vienen a solventar esto, intentando capturar y almacenar aquellos valores que sean relevantes para la red a lo largo del tiempo.

Pasamos ahora con la estructura interna de la célula de memoria:



Célula de memoria

Este es la distribución interna de una célula de memoria en una red LSTM, nos vamos a permitir un momento para repasar su "modus operandi" y como esta consigue guardar y calcular los nuevos estados en base a los inputs que obtiene.

En primer lugar, tenemos el “estado candidato”, que es aquel que se obtiene de la entrada de la salida de la célula de memoria anterior, y el input actual de esta célula, todo ello lo multiplicamos por la tanh. Esto se puede ver gráficamente, siguiendo la línea inferior y tomando la tercera bifurcación que sube donde se aplica la multiplicación de la tanh.

$$N_t = tanh[U_c * X_t + V_c * h_{t-1} + b_c]$$

Estado candidato

En segundo lugar, tenemos la función que se encarga de decidir aquello que debemos olvidar o conservar con respecto al estado de la célula anterior. Esta sigue un procedimiento similar al anterior, no obstante, ahora utilizamos la primera bifurcación donde aplicamos la multiplicación por la sigmoide.

$$f_t = \sigma[U_t * X_t + V_f * h_{t-1} + b_t]$$

Estado anterior de la célula

Ahora tenemos el homónimo de la función anterior, esta función nos va a indicar que tenemos que olvidar o mantener con respecto al nuevo estado candidato calculado en la primera función. Para este caso, solamente utilizamos el segundo “camino”, el cual es que único que nos quedaba pro explorar.

$$i_t = \sigma[U_i * X_t + V_i * h_{t-1} + b_i]$$

Nuevo estado candidato

A continuación, procedemos a calcular el nuevo estado de la célula, que como veníamos adelantando, no deja de ser una combinación de las funciones anteriores. El nuevo estado va a ser la suma de aquello que debemos mantener del estado anterior y lo que debemos mantener des este nuevo estado candidato. O lo que podría representarse, como la multiplicación del estado de la célula anterior y la función “f(t)” sumado al nuevo estado candidato multiplicado por “i(t)”.

$$C_t = f_t * C_{t-1} + i_t * N_t$$

Nuevo estado de la célula

Con nuestro nuevo estado ya calculado, vamos a proceder a la obtención de las dos salidas de la célula, esta primera salida “o(t)” será con la que decidimos que pasar de nuestro estado actual a la siguiente célula.

$$o_t = \sigma[U_o * X_t + V_o * h_{t-1} + b_o]$$

Salida Ct

Esta última función, nos indica la salida de la célula. Que se basa en el estado de esta y la función previa para determinar la información que utilizamos en el siguiente estado.

$$h_t = o_t * tanh[C_t]$$

Salida

Implementación en Python:

La implementación realizada para este modelo en Python sigue la línea de lo visto anteriormente en nuestro perceptrón. Solamente tenemos que realizar modificaciones en el apartado que respecta a la implementación del modelo secuencial:

```
model = Sequential()

model.add(Input(shape = (n_steps, n_features)))
model.add(LSTM(128, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(64))
model.add(Dropout(0.2))
model.add(Dense(1))
```

Seguimos utilizando la librería "keras" para la implementación de LSTM, en este caso vemos como simplemente realizamos un cambio en las capas que añadimos al modelo. Nuestro primer acercamiento a las predicciones va a ser con la siguiente arquitectura, la cual indagaremos con más detalle en el próximo punto.

Respecto a la preparación de los datos y tratamiento del dataset, seguimos manteniendo exactamente la misma estructura que se ha explicado en el capítulo anterior, solo se ha modificado la parte relativa al modelo.

Nuestro dataset:

Tras la parte teórica ya mencionada, nos cabe esperar una mejora sustancial con respecto al MLP, debido a que estas capas LSTM son especialistas en el tratamiento de series temporales y captación de patrones a lo largo de una secuencia. Para situar el rendimiento de este nuevo modelo y ponernos en situación, vamos a observar los resultados obtenidos para 100, 200 y 500 épocas:

100->

```
LSTM: [1248.6] 1520.6, 1291.6, 1378.5, 1366.5, 1467.9, 1265.1, 1295.7
```

200->

```
LSTM: [1054.0] 1270.7, 1046.3, 1131.0, 1129.0, 1236.7, 1065.9, 1088.1
```

500->

```
LSTM: [566.5] 621.9, 533.7, 678.6, 697.7, 615.4, 573.2, 527.3
```

2000->

```
LSTM: [474.7] 465.2, 537.1, 663.8, 560.7, 378.1, 419.3, 371.9
```

Si ilustramos el conjunto de errores obtenidos en cada sección de entrenamiento, cambiando el número de épocas. Obtenemos lo siguiente:

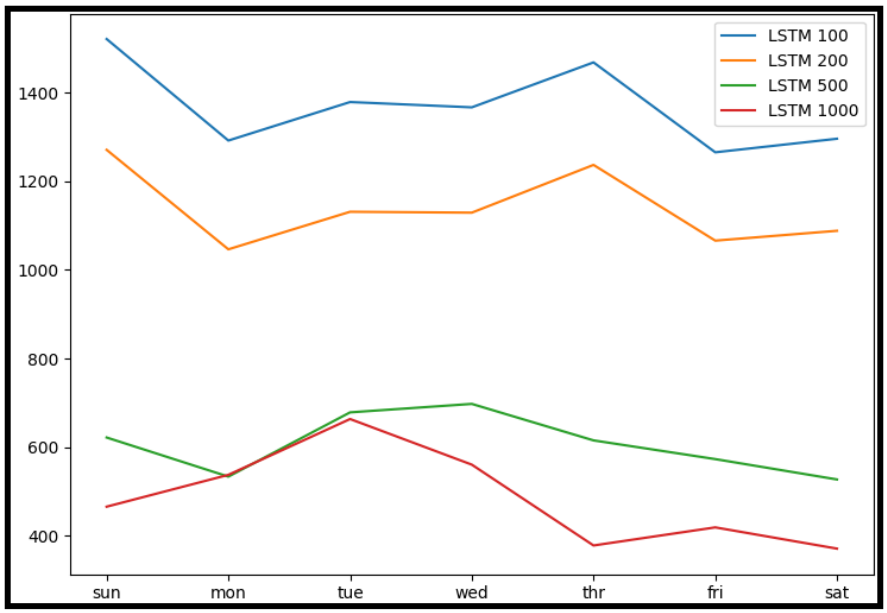
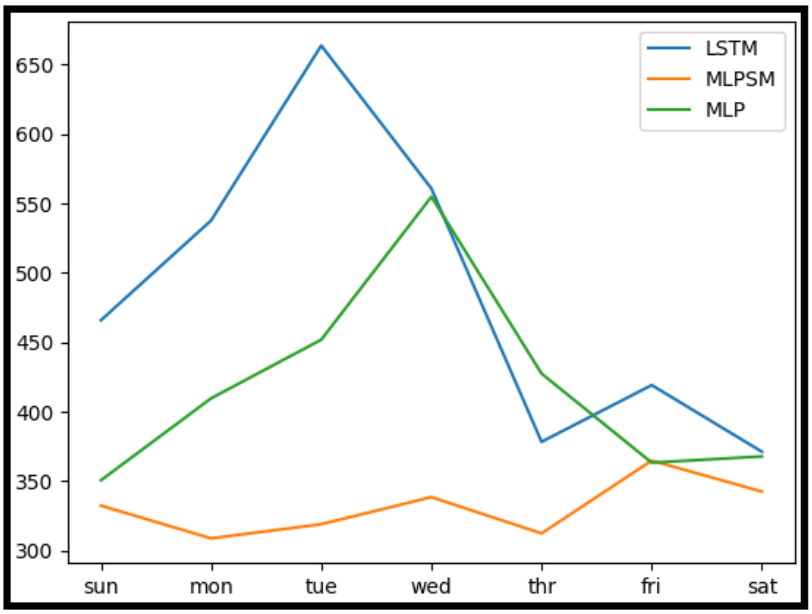


Ilustración de las distintas étapas de entrenamiento

En las primeras iteraciones, cuando la red todavía solo ha tenido entre cien y doscientas épocas para entrenar, no está estable y se refleja en el rápido descenso que sufre en cuanto damos algunas épocas adicionales. Posteriormente vemos como en mil, tiene mejores resultados que cuando solo se entrena quinientas veces, por último, hemos probado a realizar sesiones de entrenamiento más prolongadas y los resultados se quedan estables con respecto a las ya presentadas.

Ahora echando la vista atrás y vislumbrando una comparación directa con los modelos ya vistos en esta memoria, obtenemos lo siguiente:



Comparación LSTM MLPSM MLP

Comparándolo con el univariate y el multivariate multistep, vemos que este último si es bastante superior a nuestro actual modelo LSTM, pero que este no se queda lejos del ya presentado "univariate".

Tras realizar entrenamientos con distintas configuraciones del modelo, variando el número de capas LSTM, y añadiendo capas Densas, no obtenemos mejores resultados que los ya comentados. Uno de los utilizados es el siguiente:

```
model = Sequential()
model.add(LSTM(64, return_sequences=True, input_shape=(n_steps, n_features)))
model.add(Dropout(0.2))
model.add(LSTM(32, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(16))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))
```

Estos resultados se pueden deber al conjunto de datos, hasta este punto se ha utilizado un conjunto reducido que trata los datos a nivel diario y no a nivel horario dentro del día. Es por eso por lo que en primera instancia creemos que, si comenzamos a tratar ese conjunto de datos, encontraremos mejoras significativas ya que habrá muchísima más información a tratar y muchas más dependencias de las que la red puede aprovecharse. Sin embargo, para este punto nos quedamos con el siguiente resultado, el cual avala lo comentado hasta este punto:

```
LSTM: [515.6] 557.4, 555.5, 737.6, 627.9, 454.2, 395.3, 353.9,
```

CNN:

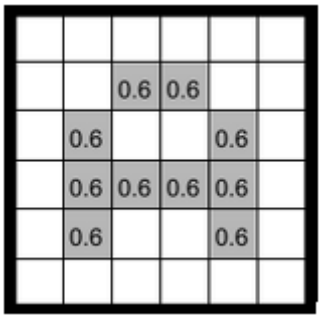
Redes Neuronales Convolucionales o Convolutivas, son el nombre de la arquitectura que da nombre a este apartado de proyecto. En este caso vamos a tratar un nuevo enfoque, que, si bien no se especializa en el análisis de series temporales, si es interesante su uso en el estudio de estas. Las redes CNN destacan en los avances que han supuesto para el reto de "la visión por ordenador", ya que estas se especializan en el procesamiento de imágenes, en la capacidad de detectar patrones y analizarlos.

Para ello, nuestra explicación se apoyará en la teoría que se utiliza para el procesamiento de imágenes para comprender los fundamentos que vamos a manejar en este punto, y posteriormente extrapolarlos a nuestro problema de predicción de series temporales.

El objetivo de una red convolucional, en el caso del procesamiento de imágenes, el de simular el comportamiento del procesamiento humano. Intenta al ver la siguiente imagen determinar que se trata de la letra "A".

Imagen entrada de la red

¿Y cómo consigue eso? Obteniendo patrones de la imagen y procesándolos. Somos capaces de identificar que ese conjunto de líneas interconectadas hace referencia a la letra "A" debido a que hay dos en verticales, una superior que las une y una intermedia a media altura.

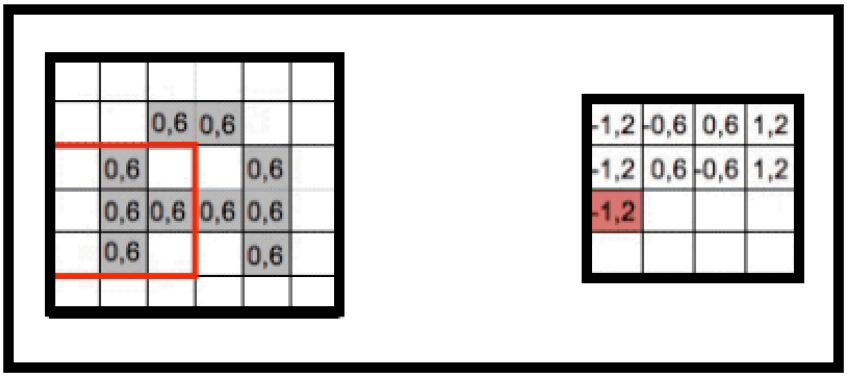


Valores de los pixeles

Refiriéndonos a la red, esta tomará como entrada una imagen, en este caso nuestra figura anterior. La cual tiene unas dimensiones de 6x6, que desemboca en un total de 36 neuronas. Si la imagen fuera a color, sería necesario realizar este tratamiento 3 veces, ya que dividiríamos la imagen en los tres canales, rojo, verde y azul. No obstante, para nuestra comprensión e implementación no indagaremos en este apartado.

El orden importa, y es que no es lo mismo que el tercer píxel tenga el valor 0.6 a que lo tengan el cuarto o el quinto. Esto es algo positivo para nosotros, las series temporales también comparten esta característica, la capacidad de tener en cuenta el orden es vital.

Convoluciones, el proceso que da nombre a este tipo de red neuronal. Este proceso se fundamenta en la aplicación de un filtro, al que denominaremos *kernel*, e ir aplicando el producto escalar en agrupaciones de píxeles cercanos entre sí.



Procesamiento del kernel

Esta figura representa ese proceso, donde vemos una matriz 3x3, que va aplicando esta operación matemática a cada grupo de pixeles cercanos para al final desembocar una nueva matriz bidimensional.

Una pregunta que nos puede surgir llegados a este punto es, ¿qué valores le damos al filtro? La respuesta nos la tendrá que dar nuestra red, ya que será la encargada de ajustar esa parte.

Este proceso no se queda únicamente en la utilización de un filtro, sino que disponemos de "X" y cada uno nos dará una nueva imagen como vemos en la figura anterior. Cada imagen tendrá como objetivo captar una característica distinta de la imagen, que permita realizar las distinciones que necesitamos.

Para ilustrar este ejemplo hemos utilizado figuras, como la anterior, donde se ilustran capas Conv2D, una imagen. No obstante, para nuestro fin es necesario que sean unidimensionales ya que la entrada será una serie temporal. El funcionamiento de esta capa es exactamente igual, no obstante, la entrada será un array de valores con los puntos de la serie temporal, en este caso. Los filtros del kernel siguen siendo una matriz la cual realiza el mismo proceso que en el ejemplo anterior.

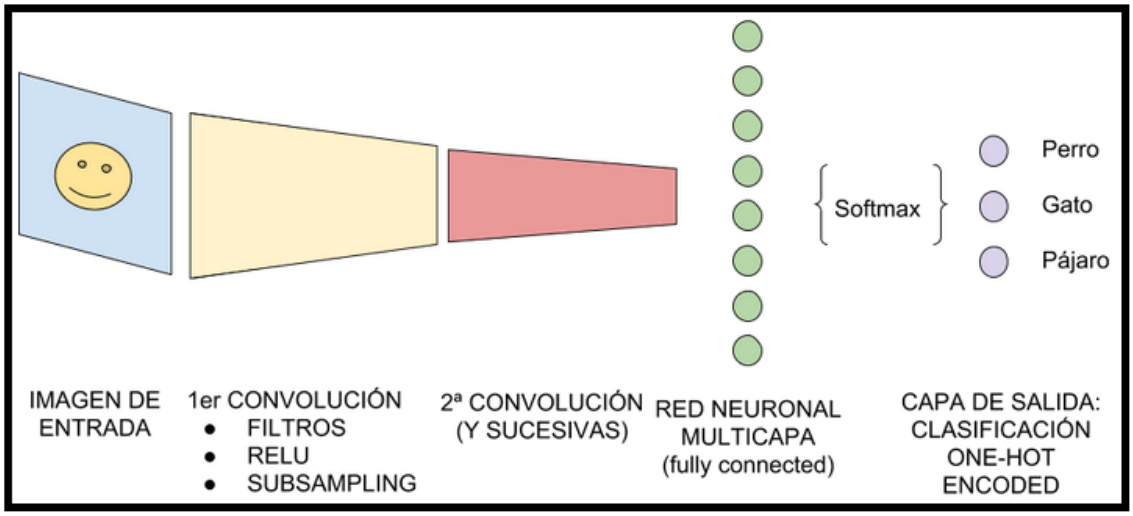
Nos interesa disminuir el número de neuronas mediante la técnica llamada Subsampling, proceso por el cual reducimos la cantidad de neuronal antes de hacer una nueva convolución. ¿Por qué es necesario reducir nuestro número de neuronas? Lo normal sería pensar, que este proceso no es necesario y que incluso podría desembocar en la perdida de información de nuestra red. Para entender esta necesidad, vamos a contar el número de neuronas que disponemos para nuestra imagen anterior aplicando un total de 32 filtros, sería del orden 6x6x32, 1152 neuronas para una imagen de 6x6 y sin escala de colores. Esto es únicamente tras la primera convolución, lo que deja en evidencia que es necesario reducir el número de neuronas. El subsampling, se basa en la

agrupación de píxeles cercanos para su posterior agrupación, prevaleciendo las características principales que detectó ese filtro.

Tras realizar este proceso obtenemos una salida de esta primera capa de convolución. Esta no es más que la primera piedra en este proceso, ya que esta capa solo es capaz de vislumbrar patrones sencillos como líneas o curvas, no obstante, a media que vayamos aplicando más y más convoluciones, la red será capaz de detectar patrones más complejos, hasta llegar a "ver".

Tras este procesamiento, la salida de la última capa convolucional desemboca en una red multicapa tradicional, donde normalmente se utiliza a modo de clasificador, pero para nuestro caso utilizaremos una red neuronal para realizar una predicción.

A modo de resumen, debemos tener clara la idea de que estas redes funcionan como un "embudo", entran datos de gran tamaño, de los cuales vamos a intentar aprender sus principales características y patrones para irlos reduciendo hasta que sean tratados por una red neuronal multicapa.



Red Neuronal Convolucional

Implementación en Python:

La siguiente implementación, sigue el mismo esquema ya estudiado en el código general, en este caso, aplicando convoluciones de una dimensión. Cuando entramos en el modelo a entrenar, disponemos de lo siguiente:

```
model = Sequential()

model.add(layers.Conv1D(32, 3, activation='relu', input_shape=(n_steps, n_features)))
model.add(layers.MaxPooling1D(2))
model.add(layers.Conv1D(64, 3, activation='relu'))
model.add(layers.MaxPooling1D(2))
model.add(layers.Conv1D(64, 3, activation='relu'))
model.add(LSTM(64))
model.add(Dropout(0.2))
model.add(Dense(32, activation='relu'))
model.add(Dense(1))
```

La red dispone de una capa de entrada convolutiva de una dimensión, donde tenemos los "n_steps" y "n_features" que conforman los datos de entrada. Posteriormente un "MaxPooling", repetimos esta estructura otra vez y posteriormente aplicamos una capa LSTM a la extracción de características realizadas por las capas anterior. Terminamos con una capa de "dropout", para evitar sobreajuste, y "Dense" para el procesamiento final, junto a una única salida, ya que solo vamos a predecir 1 valor.

Nuestro dataset:

Ahora procederemos a realizar la comparación del modelo según la variación de las épocas de entrenamiento, para ver cual es la franja óptima en la que nos quedaremos, posteriormente echaremos un vistazo atrás para compararlo con el modelo LSTM anterior.

Épocas 100->

```
CNN: [492.1] 494.7, 409.1, 491.0, 519.1, 525.2, 475.3, 324.7,
```

Épocas 200->

```
CNN: [531.0] 441.5, 487.1, 557.3, 545.4, 518.5, 464.6, 291.9,
```

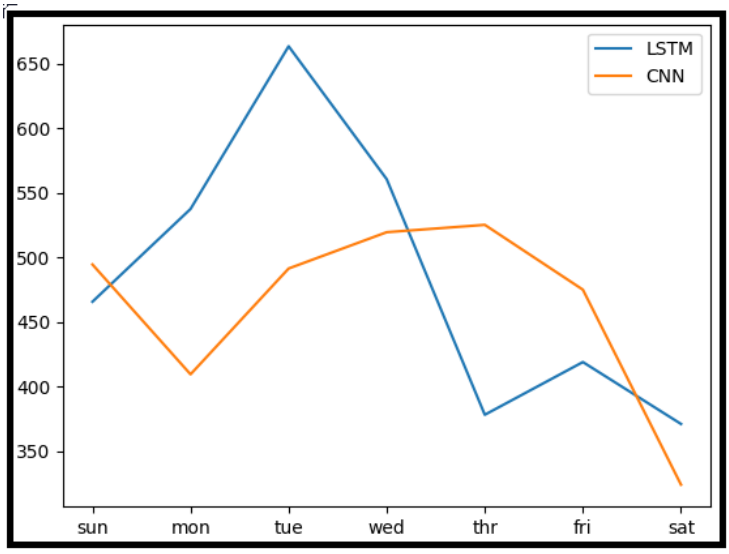
Épocas 500->

```
CNN: [536.4] 442.2, 490.6, 560.0, 547.7, 531.5, 465.2, 295.9
```

Épocas 2000->

```
CNN: [538.6] 445.2, 493.9, 562.6, 549.8, 524.8, 467.7, 288.0
```

Vemos errores similares a los estudiados en los modelos LSTM, rondando los mismos valores. También podemos observar, como la red se estabiliza a las 100 épocas, y los resultados de seguir entrenando la red tienden a ser contraproducentes, en tiempo y calidad de estos. Con esto podemos determinar que para el dataset que estamos tratando, tanto el modelo LSTM como CNN, aunque este último tenga un pequeño tratamiento mediante una capa LSTM, rinden de manera similar para los datos que queremos predecir.



Comparativa LSTM y CNN

La CNN, como muestra la gráfica, obtiene unos registros algo menores. Ya que recordemos, esta gráfica indica el error cuadrático medio por día producido en la predicción, la diferencia global es de 10, que en relación con la magnitud de valores que manejamos en el dataset, es insuficiente para decantar la balanza a un lado u al otro.

Multihead:

El modelo multihead, aplicado al perceptrón multicapa, fue uno de los que mejores resultados arrojó en el estudio de los distintos modelos posibles para el perceptrón. En esta ocasión vamos a seguir el estudio aplicando su principio, el disponer de dos entradas distintas que luego convergen para dar una salida única en la red.

En este caso hemos decidido repartir los conjuntos de variables por las dos ramas, en la primera tenemos los parámetros los parámetros “conocidos”, como el voltaje, mientras que en la segunda entrada y rama del multihead disponemos del resto de features numeradas.

Sería interesante realizar un estudio del comportamiento del modelo para las diferentes configuraciones de entrada en la red, modificar para que entrara la variable objetivo únicamente por una entrada, mientras que en la otra disponemos del resto de datos de nuestro dataset.

Para esta ocasión, hemos optado tras diversas comprobaciones, utilizar únicamente capas “Densas” para la composición de la red, en ambas ramas se tiene una distribución similar, para posteriormente realizar una combinación para obtener la predicción.

Implementación en Python:

```
input_layer_1 = Input(shape=(n_steps, n_features1))
input_layer_2 = Input(shape=(n_steps, n_features2))

side_x = Embedding(n_features+1, 32, input_length=1)(input_layer_1)
side_x = Reshape((-1,))(side_x)
side_x = Dense(64, activation='swish')(side_x)
side_x = Dropout(0.1)(side_x)
side_x = Dense(64, activation='swish')(side_x)
side_x = Dropout(0.1)(side_x)
side_x = Dense(64, activation='swish')(side_x)
side_x = Dropout(0.1)(side_x)
# Rama de features
side_y = Reshape((-1,))(input_layer_2)
side_y = Dense(256, activation='swish')(side_y)
side_y = Dropout(0.1)(side_y)
side_y = Dense(256, activation='swish')(side_y)
side_y = Dropout(0.1)(side_y)
side_y = Dense(256, activation='swish')(side_y)
side_y = Dropout(0.1)(side_y)

x = Concatenate(axis=1)([side_x, side_y])

x = Dense(512, activation='swish', kernel_regularizer="l2")(x)
x = Dropout(0.1)(x)
x = Dense(128, activation='swish', kernel_regularizer="l2")(x)
x = Dropout(0.1)(x)
x = Dense(32, activation='swish', kernel_regularizer="l2")(x)
x = Dropout(0.1)(x)

output = Dense(1, name='output')(x)
model = Model(inputs = [input_layer_1, input_layer_2],outputs = [output])
```

Implementación del modelo MultiHead

Nuestro dataset:

Nuestro dataset en este caso es algo limitado en cuanto a dimensionalidad para la magnitud de la red que queremos entrenar, no es posible realizarlo con nuestro dataset utilizado para el resto de los modelos. Esto de debe a que este carece del número suficiente de muestras para poder entrenar a la red a lo largo del tiempo sin que sufra sobre entrenamiento. Para solventar este contratiempo hemos utilizado un dataset derivado del nuestro con una agrupación temporal de los datos en minutos y no en días, lo que ayuda en gran medida a solventar el problema que estamos comentando.

Por otro lado, a la hora de realizar la predicción nos aparece otro contratiempo que surge cuando queremos realizar la comparación de estos datos con los anteriores modelos. Cuando se presentan los datos diarios, estos se dan como la suma de los valores diarios recogidos en un hogar, y tienen la magnitud de la suma de todo el día, mientras que cuando realizamos la predicción por minutos, obtenemos unos valores mucho más pequeños, ya que aquí vamos a predecir fracciones de tiempo mucho menor.

Comparaciones entre todos los modelos:

Como antesala, vamos a recordar que el motivo de realización de este proyecto es ahondar en los motivos por lo que la predicción de series temporales sigue siendo o no la última barrera para el Deep Learning. Nuestra motivación radica en la comprensión y entendimiento de los

resultados y comportamientos obtenidos en los distintos modelos para nuestro conjunto de datos. Con esto dicho, pasamos a la comparación de los distintos modelos.

Ahora comenzaremos una comparación de los modelos clásicos y estos modelos basados en el Deep Learning, como son los basados en el Perceptron multicapa, CNN y LSTM. Para ello vamos a intentar tener en cuenta no solo los resultados numéricos relativos al error de estos, sino también la complejidad computacional de los mismos derivados del tiempo que necesitamos para realizar una predicción. En primer lugar, estudiaremos un concepto clave para entender por qué actualmente los modelos basados en aprendizaje profundo tienen una barrera frente a los modelos clásicos estadísticos.

Ergodicidad, es la palabra clave para entender la "barrera" de la que hablamos anteriormente. Como definición en el contexto que nos atañe, la ergodicidad, significa que las características y propiedades promedio del proceso se mantienen constantes a lo largo del tiempo y pueden ser estimadas a partir de una única muestra suficientemente larga del proceso.

Volviendo al concepto que nos presentan las series temporales y su predicción, en este punto nos referimos a qué momentos muestrales calculados a partir de una serie temporal tienden a converger en algún sentido con un número finito de muestras. Si esto no se da, a efectos tendríamos una serie que nunca converge, con lo que nuestros modelos de aprendizaje profundo nunca podrán realizar una predicción acertada.

Necesitamos disponer de una convergencia para poder asumir que los valores previos tienen una fuerte presencia en el devenir de la serie, de lo contrario, se podría tratar de un proceso aleatorio, que nunca..

La serie que hemos tratado, aunque no se pueda demostrar matemáticamente tiene todas las condiciones para cumplir el supuesto de ergodicidad, lo que nos avala los resultados que hemos obtenido, ya que, de lo contrario, estos hubieran sido inconsistentes.

Un concepto relacionado con la ergodicidad es el camino aleatorio o random walk… Aunque no podamos demostrar matemáticamente si estamos o no ante un camio aleatorio, podemos aplicar una prueba de estacionariedad como es el test de Dickey Fuller Augmentado (DFA), que, aunque no nos permite garantizar estar ante un camino aleatorio, sí nos permitiría en algunos casos descartarlo.

Los caminos aleatorios se caracterizan por ser una sucesión de incrementos o decrementos aleatorios en una sucesión. Esto es un problema grave para nuestra red, ya que indica que los valores anteriores no serán buenos condicionantes para la siguiente predicción de nuestra red. Si el siguiente valor es un incremento o decremento aleatorio, nuestra red no tiene manera posible de aprender esa relación, ya que a efectos no existe entre el valor anterior y el próximo.

Para ilustrar ambos casos con un ejemplo, podemos hablar de lo siguiente: temperatura diaria y precio de las acciones.

Temperatura diaria, como uno que si cumple este principio. La ergodicidad aquí se basa en la suposición de que las condiciones climáticas se mantienen estables durante un largo periodo de tiempo, y no son afectadas por cambios drásticos o eventos atípicos que la afecten de manera significativa. Si se diera el caso de que un evento externo altera la temperatura de un ecosistema, como por ejemplo la construcción de cierto complejo o estructura en un terreno, ya no se cumpliría este principio.

Por otro lado, el precio de las acciones, este caso es el más típico donde podemos decir que no se cumple el principio. El precio de una acción está afectado por una infinidad de agentes externos, noticias económicas o políticas, eventos inesperados, y un sinfín de cuestiones que afectan al valor a lo largo del tiempo, los cuales hacen que no se cumpla el principio de la ergodicidad.

Podemos concluir, cuando disponemos de una serie que se ve afectada por agentes que no permiten que la misma converja ya que la hacen dependientes de otros factores que son externos a esa variable, diremos que no cumple el principio de ergodicidad. No obstante, como hemos comentado no es posible disponer de una regla o fórmula inmutable para calcularla en estos casos.

A continuación, vamos a realizar una primera comparación entre el Univariate y el modelo ARIMA, utilizado anteriormente:

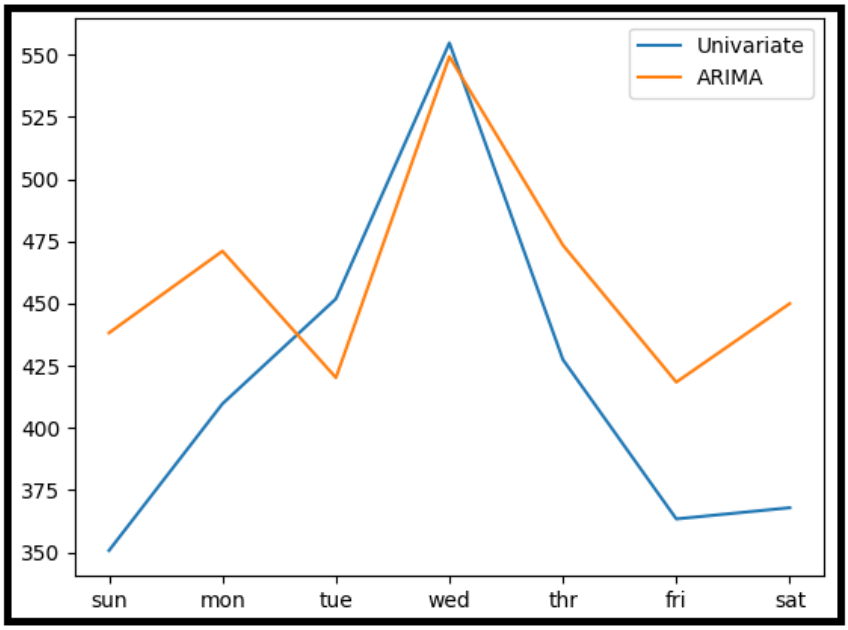


Ilustración comparativa ARIMA y Univariate

En este caso, disponemos de resultados muy similares en los modelos, no obstante, tenemos que recordar que el costo computacional y de tiempo que supone el modelo ARIMA es menor que el necesario para el realizar las predicciones en el MLP.

Si realizamos una comparación con el último modelo estudiado, el “Multivariate Multi-Step”, vemos que si se aprecia una diferencia a favor de este segundo:

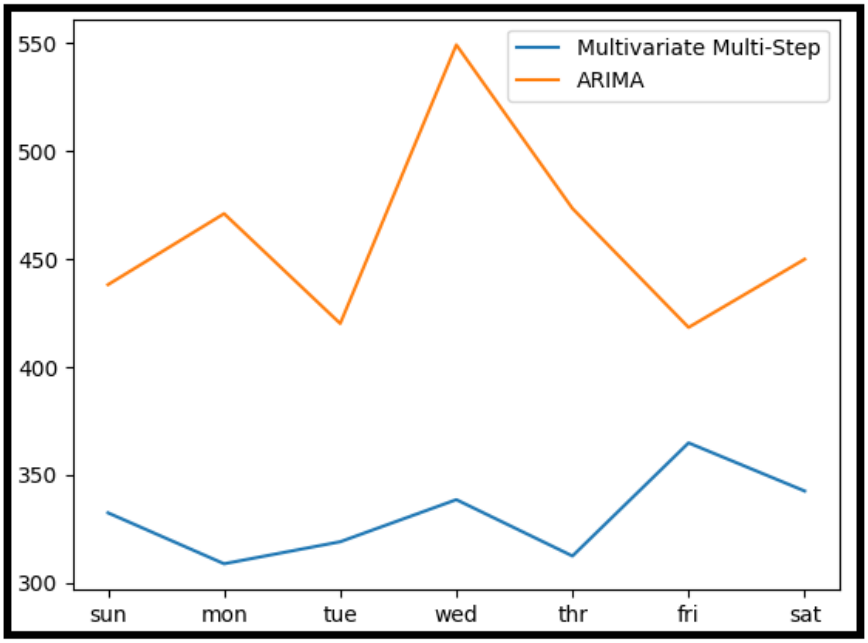


Ilustración comparativa ARIMA y Multi-Step

Volvemos a lo mismo, no es posible realizar una comparación donde se reflejen todos los agentes que influyen en el proceso, y solo podemos limitarnos a una medida de los errores que se han producido en una predicción de la red, lo que desemboca en algo injusto y que puede o no suponer un error a la hora de valorar los datos. Por nuestra parte, los tomaremos como representativos para la conclusión que queremos dar, y es si en este apartado los clásicos son mejores.

En base a este problema y consecuente necesidad, hemos indagado en busca de una solución práctica y hemos encontrado una solución. “Makridakis Competitions” son un conjunto de datasets creados por el experto en pronóstico y análisis de series temporales Spyros Makridakis. Las Makridakis Competitions, son una serie de competiciones que van desde el año 1982 hasta la actualidad, en las cuales, mediante un amplio número de series temporales, se busca tener la capacidad de evaluar las bondades y debilidades de los modelos y métodos de pronóstico en un apartado “unificado” para así poder compararlos entre sí.

No.	Informal name for competition	Year of publication of results	Number of time series used	Number of methods tested	Other features
1	M Competition or M-Competition[1][5]	1982	1001 (used a subsample of 111 for the methods where it was too difficult to run all 1001)	15 (plus 9 variations)	Not real-time
2	M-2 Competition or M2-Competition[1][6]	1993	29 (23 from collaborating companies, 6 from macroeconomic indicators)	16 (including 5 human forecasters and 11 automatic trend-based methods) plus 2 combined forecasts and 1 overall average	Real-time, many collaborating organizations, competition announced in advance
3	M-3 Competition or M3-Competition[1]	2000	3003	24	
4	M-4 Competition or M4 Competition	2020[7]	100,000	All major ML and statistical methods have been tested	First winner Slawek Smyl, Uber Technologies
5	M-5 Competition or M5 Competition	Initial results 2021, Final 2022	Around 42,000 hierarchical timeseries provided by Walmart	All major forecasting methods, including Machine and Deep Learning, and Statistical ones will be tested	First winner Accuracy Challenge: YeonJun In. First winners uncertainty Challenge: Russ Wolfinger and David Lander
6	M-6 Competition or M6 Competition	Initial results 2022, Final 2024	Real time financial forecasting competition consisting of 50 S&P500 US stocks and of 50 international ETFs	All major forecasting methods, including Machine and Deep Learning, and Statistical ones will be tested	

Makridakis Competitions

En la anterior imagen podemos ver la evolución del número de series temporales utilizadas a lo largo de los años. Para el alcance de este trabajo de fin de grado, queda excluida la realización de una comparación entre los distintos modelos utilizando estos datasets, ya que son de órdenes muy grandes y no es motivo final de este proyecto.

Transformers y las series temporales:

Desde que comenzamos a realizar este trabajo de fin de grado, una nueva oleada ha irrumpido con fuerza en el terreno de la inteligencia artificial, haciendo inviable no dedicarle un punto para ver su desempeño en la predicción de series temporales. Hablamos de los Transformers. Como apunte, cabe recalcar que los Transformers fueron creados en diciembre de 2017, como se indica más adelante, pero su auge empieza a surgir en este último año y medio que es la fecha de realización de este trabajo.

En este punto nos centraremos en ver si este tipo de modelos son efectivos o no para nuestro problema actual, y el motivo. En primer lugar, debemos tener claro como funcionan este tipo de soluciones.

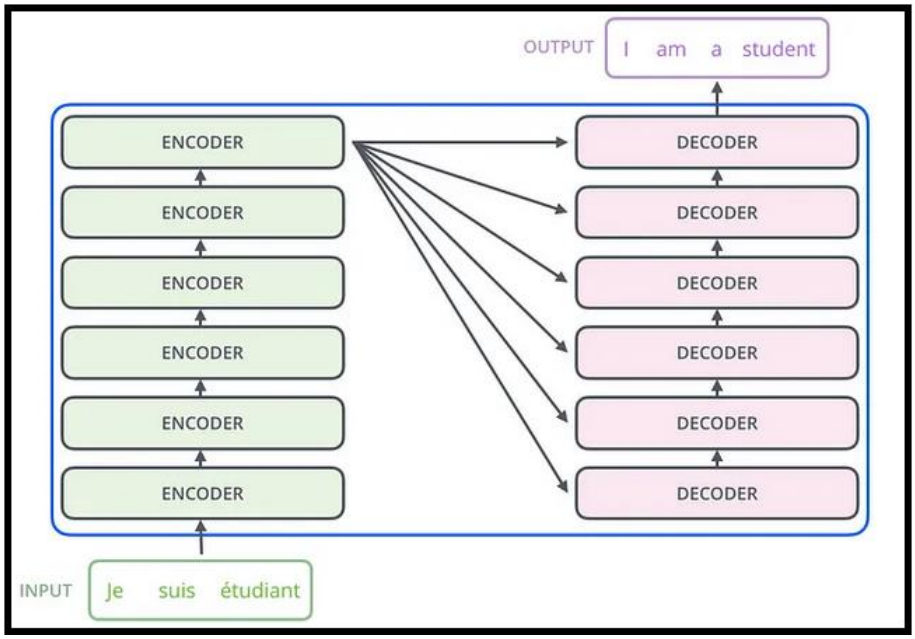
¿Cómo funciona un Transformers?

Un Transformers en primera instancia lo podemos definir como una caja negra de:

Input -> Transformers -> Output

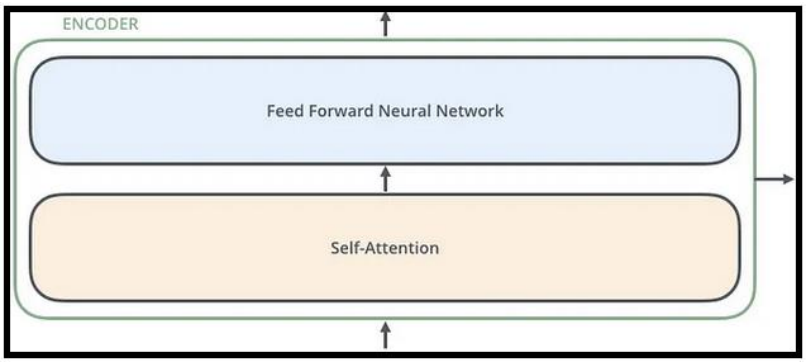
Esto es reducir a su máxima expresión lo que conocemos como Transformers. Si queremos indagar un poco más en esa "caja negra", veríamos que se trata de un conjunto de "encoders" y "decoders". Nuestro input, pasa por una serie de "encoders" que se anidan uno tras otro para luego mandar su salida a una serie de "decoders" que harán lo propio.

En la siguiente imagen vemos este esquema, donde para realizar la traducción del francés al inglés de la frase "Je suis étudiant", esta pasa por el flujo ya menciona para obtener la traducción en inglés.



Arquitectura de un transformers

No obstante, todo esto todavía sigue quedando bastante abstracto, para ello vamos a hacer otro aumento más. En esta ocasión, ya podemos observar un término crucial para dar vida a esta idea que son los transfomers, el "Self-Attention":



Encoder

Dentro de esta parte del Transformers, vemos un componente de atención y una red neuronal "corriente", como las que ya hemos estudiado.

Antes de comenzar a indagar en cada parte completa del Transformers vamos a hablar del "Embeddings", que es la piedra angular sobre la que va a empezar a trabajar este modelo.

Embeddings y Positional Encoding:

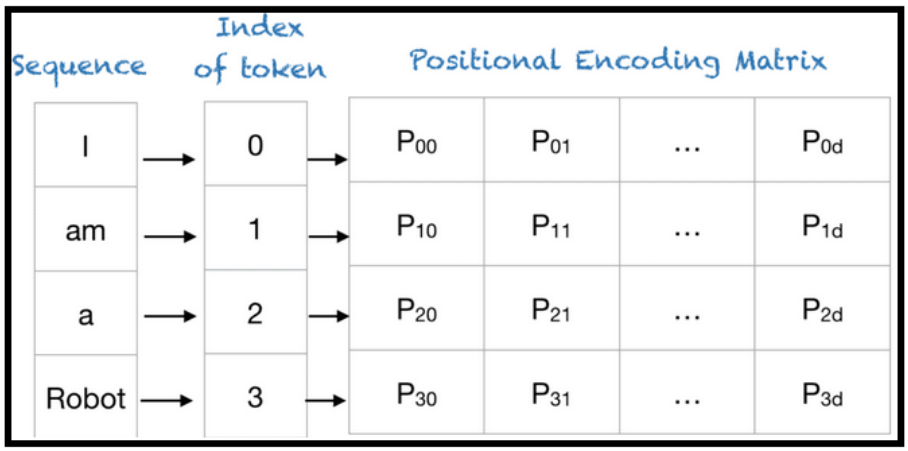
En los NLP o procesamiento de lenguaje natural, el uso de "embeddings" ya era una técnica más que conocida. En primer lugar, necesitamos convertir las palabras a un valor numérico, que denominaremos "token". Inicialmente en el paper "Attention is all you need" (https://arxiv.org/abs/1706.03762), publicado en 2017, donde se trata por primera vez este tema, se decide utilizar un vector de "embeddings" de 512 dimensiones, así que para nuestra explicación seguiremos esa pauta.

Palabra -> Token -> Vector n-dimensional (512)

Este sería el flujo que acabamos de definir. Como última nota, es conveniente saber que no siempre la relación Palabra – Token es 1-1, sino que puede haber más de un token asociado a una palabra.

El orden importa, es una frase más que relevante en este documento, y para los Transformers no lo es menos. En el caso del procesamiento de texto, no es lo mismo "Bueno, que pepe" que "Pepe, que bueno", con lo que para paliar esta necesidad se creó la siguiente técnica.

Positional Encoding, es la solución para conseguir preservar la importancia del orden en el contexto de la frase, por ejemplo. En primera instancia, se puede llegar a pensar que podría ser algo tan trivial como añadirle un índice o un identificador de secuencia al principio, pero al intentar normalizarlo entre 0 y 1, puede presentar problemas para longitud de secuencias variables, y un largo etcétera.



Matrix del "Positional Encoding"

Para ello se creó la imagen previa, donde observamos una matriz para representar la posición de cada elemento. Para ello, el valor que se encuentra en cada posición va a venir dado por las siguientes fórmulas:

$$P(k, 2i) = \sin\left(\frac{k}{n^{2i/d}}\right)$$

$$P(k, 2i+1) = \cos\left(\frac{k}{n^{2i/d}}\right)$$

Fórmula para obtener los valores de la matriz P

k: posición del valor.

d: dimensión final de la matriz.

P(k,j): Posición del elemento

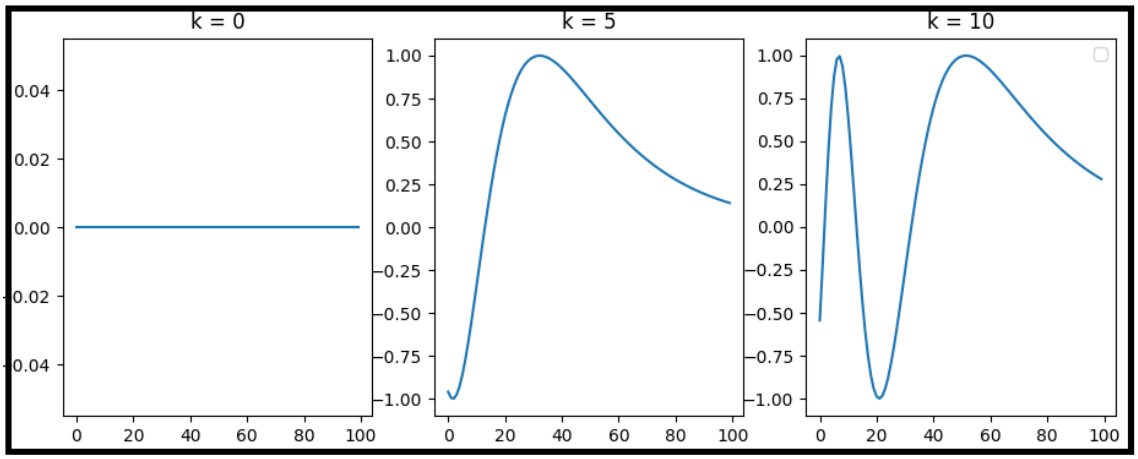
n: "User-defnied" scalar, utilizaremos el valor 10.000*

i: Valor de las columnas, 0 <= i < d/2

(*) Utilizaremos este valor, que viene dado en el paper "Attention Is All You Need".

Se utilizan funciones trigonométricas, dada su naturaleza periódica y moldeable. Como recordamos la función senoidal tiene valores entre [-1, +1]. Y los parámetros que podemos modificar, son la frecuencia que es el número de ciclos completados por segundo y la longitud de onda, que es la distancia entre cada repetición.

Con todo esto conseguimos valores normalizados entre un rango y dos parámetros los cuales se pueden modificar para representar cada valor. Para el ejemplo que vemos a continuación, no hemos calculado toda la matriz, sino que hemos hecho la representación de la función para tres valores ejemplo y así ver su comportamiento.



Representación de la función para los distintos valores de k

Los conceptos que tenemos que extraer de este punto, es que los Transformers codifican sus palabras en vectores n-dimensionales y tienen un mecanismo para poder tener en cuenta la posición.

Self-Attention:

Este mecanismo consiste en dar fuerza a las relaciones entre dos inputs del "Transformer", por ejemplo, en dos palabras de una oración. Esta relación es crucial para el procesamiento de texto, por ejemplo:

"Mi perro duerme cálido en el salón"

"Mi perro duerme en un salón cálido"

Vemos como la misma palabra cálido, en una oración hace referencia al perro y en la otra al salón, para detectar estas relaciones y darles peso se creó el mecanismo de atención.

Para conseguir esto, se utiliza un "Soft Dictionary". Que es un almacenamiento donde guardamos la "atención" entre dos palabras, por ejemplo, en la primera frase la relación de "perro vs cálido" deberá ser 1, indicando mucha relación. Mientras que en la segunda deberá ser de -1, ya que cálido se refiere al salón.

Para poder calcular este valor, se utilizan 3 matrices llamadas "Query-Key-Value":

Q: tokens a evaluar.

K: tokens a evaluar como claves del diccionario.

V: tokens de salida.

Para ilustrar esta parte con un ejemplo, para nuestra frase tendríamos lo siguiente:

"perro": [0.4, -0.1, 0.5, ...]

"cálido": [0.3, -0.4, 0.2, ...]

(*) Esto se realiza con todas las palabras, pero para ilustrar el ejemplo vamos a centrarnos en la relación "perro" vs "cálido".

Ahora dividimos los vectores en Q K V:

Q: "perro": [0.4, -0.1, 0.5, ...]

K: "cálido": [0.3, -0.4, 0.2, ...]

V: "cálido": [0.3, -0.4, 0.2, ...]

Procedemos con el cálculo de los vectores Q y K, con algún método matemático como puede ser el producto escalar. Cuanto mayor sea la similitud entre ellos, mayor atención asignará, por ejemplo: 0.7 en este caso.

$$Z = Softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Fórmula para calcular la atención

Sin entrar en detalle, con la fórmula podemos extraer que nuestro valor de atención “Z” será la multiplicación matricial de Q por la traspuesta de K; y eso multiplicado por V. La razón de esto viene dada porque Q y K son los valores de los embeddings, nuestro token con vector n-dimensional. Como consecuente al multiplicarlos obtenemos la “similitud” entre dichos vectores. Si el embedding estuviera funcionando correctamente palabras relacionadas, como “labrador” y “dálmata”, tienen que estar más relacionadas que “labrador” y “tubería”.

Matriz de atención

Esta matrizpermite visualizar lo comentado en el párrafo anterior, vemos como “tranquilo”, que es una palabra que podría también hacer referencia a “salón” en otro contexto, aquí claramente está vinculada con la palabra perro.

Como ya estamos familiarizados con la terminología “Multi-Head”, añadir que el proceso que se realiza aquí es subdividir las 512 dimensiones del embedding en 8 grupos, sugeridos por el paper, para luego calcular la atención de cada grupo, lo que en el artículo denominan “Multi Head Attention” Tras realizar este cálculo se realiza un promedio de los resultados obtenidos por cada “head”.

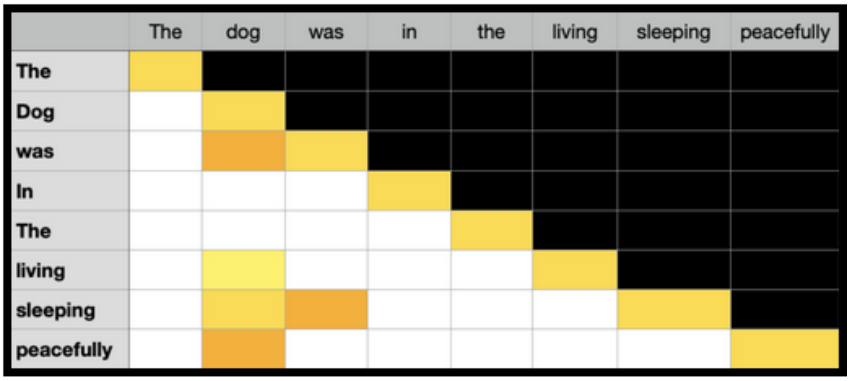
Para finalizar el apartado de la atención, debemos mencionar los tres tipos de atención de un “Transformer”:

-Self Attention

-Cross Attention

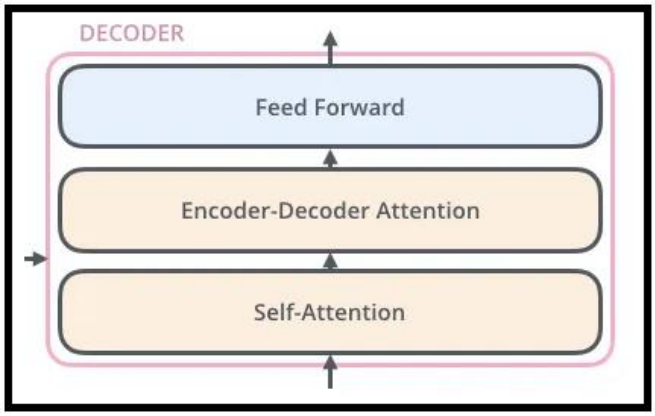
-Masked Attention

Para quedarnos con la idea principal de esta, ya que las dos siguientes no son más que modificaciones de lo mencionado anteriormente, debemos hacer hincapié en que la “Cross Attention” es que aquella que utiliza como entrada en el “decoder” el output del “encoder”. Por otro lado, la “Masked Attention”, es una técnica utilizada para no darle información futura a la red, con esto buscamos seguir el esquema natural del lenguaje en este caso. Por ejemplo, si quisiéramos traducir la frase “Mi padre estaba allí” -> “My dad was there”, la salida irá se irá procesando de manera secuencial, es decir, en primera instancia no necesito saber calcular la atención de “padre” con “allí”, ya que es adelantar información futura a la red.



Matriz Masked Attention

Con esto lo que buscamos es prevenir la obtención de valores "futuros" en relación con la atención de los tokens. Con esto concluimos la parte del "encoder". A continuación, vamos a revisar los "decoders" antes de dar por finalizada esta fase introductoria.



Decoder

El esquema es similar a lo ya visto en el apartado anterior, la principal diferencia viene dada por la "Cross Attention", ya comentada anteriormente. Una función característica de este apartado, es la utilización de la salida de la red como entrada como parte del aprendizaje.

Entonces, nuestra salida será también la entrada del "decoder" hacia los embebings, y el resto de la secuencia terminando en la "Masked Self Attention" para Q, K, V. Por último, se suma y normaliza para la entrada en la "Feed Forward".

Por último, la salida del modelo es algo tan trivial como una última capa Lineal a la cual aplicamos Softmax, para obtener un valor entre 0 y 1.

Antes de finalizar con la explicación de la arquitectura, cabe destacar que las arquitecturas basadas en "Transformers", no tienen que seguir este esquema "decoder-enconder", por ejemplo, la arquitectura GPT3, utilizada en chatgpt utiliza solamente la parte "decoder", es decir va sin encoder y todo lo que ello conlleva. El motivo detrás de esto radica en la naturaleza y objetivo de la red, sin entrar en excesivos detalles, GPT-3 tiene como objetivo principal general texto coherente y contextualmente correcto en base a una entrada proporcionada por el usuario, no es como nuestro anterior ejemplo donde tenemos como objetivo realizar una traducción de la entrada proporcionada.

Tras este breve paréntesis, volvemos al tema principal de este trabajo fin de grado, que retomamos con la siguiente pregunta:

¿Son buenos para la predicción de series temporales?

Ahora que ya sabemos las bases sobre las cuales se cimienta este nuevo paradigma de la inteligencia artificial, podemos empezar a comentar lo siguiente:

-Tienen la capacidad de mantener la temporalidad de los datos mediante el uso del "Positional Encoding".

-Tienen la capacidad de prestar atención a las dependencias fuertes entre valores, en base al mecanismo de "Self-Attention".

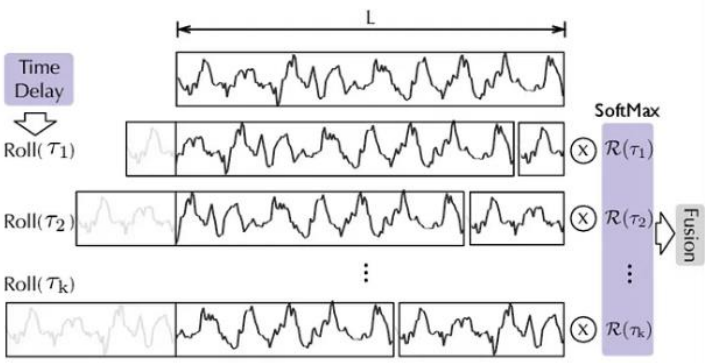
Sobre estas afirmaciones, todo parece indicar que esta arquitectura puede resultar de utilidad para el problema que estamos estudiando. No obstante, el paper "**Are Transformers Effective for Time Series Forecasting**?" publicado en 26 de mayo de 2022, indica todo lo contrario. Sustenta que, si es verdad, que el "Positional Encoding" y el uso de tokens facilita la preservación de cierta información ordenada, los mecanismos de atención rompen con esta preservación. Para validar dicha información se realizó un estudio con modelos denominados "LTSF-Lineal", modelos lineales de una única capa contra modelos "LTSF" basados en "Transformers" y en todos los casos los primeros modelos mejoraban a los segundos.

Sin embargo, el pasado 16 de junio, se publicó un artículo **(Yes, Transformers are Effective for Time Series Forecasting (+ Autoformer)),** donde se indicaba que la variante de los "Autoformers" sí son capaces de rendir mejor en la predicción de series temporales.

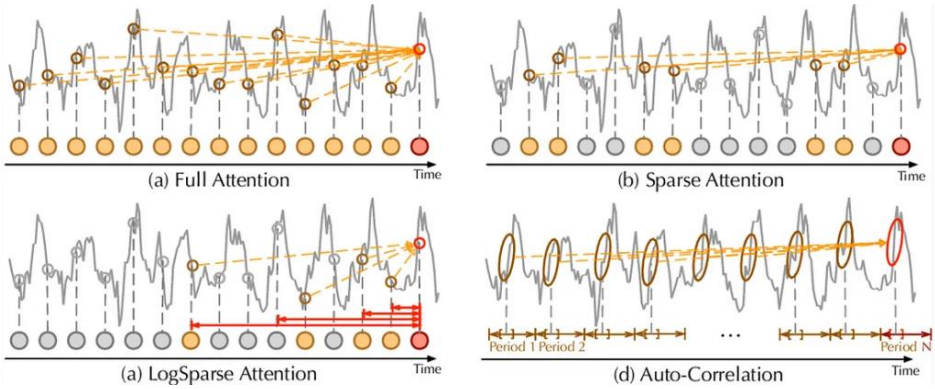
Para situar las ideas, la principal diferencia que presenta esta variante de los Transformers viene dada por los mecanismos de "Descomposition" y "Auto-Correlation", que vienen a subsanar los problemas presentados en el apartado anterior. Por orden sería lo siguiente:

Descomposition-Based Approach, consiste en "dividir" nuestra serie temporal en subseries en base a la descomposición por componentes, cada una representando una de las categorías más previsibles, por ejemplo, la parte estacional y la parte cíclica de la misma.

Auto-Correlation, con esto buscamos la relación entre el valor actual y sus valores anteriores. Esta parte viene a sustituir al mecanimos de "Self-Attention". Para realizar el cálculo del valor de "Auto-Correlation", nos ayudamos añadiendo un "delay" previo a la serie, como se observa en la siguiente figura:



Por último, vamos a ilustrar las diferencias gráficas entre el mecanismo de "Self-Attention" y "Auto-Correlation". La clave que marca la diferencia es el dominio de los datos, los modelos basados en atención son puntos seleccionados en el dominio del tiempo, dependiendo del método como se puede observar en las figuras a), b) y c) puede variar que punto utilizar o no. Sin embargo, en "Auto-Correlation" se centra en la conexión entre dos subseries en base a la división de periodos, figura d)



Como punto final, vemos que los modelos "Autoformers", sí se presentan como una solución válida para la predicción de series temporales, y que a diferencia de su variante más conocida "Transformers", si obtienen resultados que los hacen competitivos frente a los modelos clásicos.

Conclusiones:

La predicción de series temporales son un campo en continua evolución. Con el auge de nuevas técnicas y métodos para poder afrontar este paradigma, nos hace pensar que los modelos clásicos se quedan atrás y no pueden seguir el ritmo de las nuevas vertientes.

En este trabajo de fin de grado, hemos dedicado nuestro tiempo al estudio de los modelos más capaces para realizar esta operativa. Empezando por los modelos clásicos, que son los pilares y métodos a batir en este trabajo. Seguidos del Perceptrón y todas sus variantes, adentrándonos en la parte del Deep Learning, en el apartado del Deep Learning también hemos comentado acercamientos más potentes como son LSTM y CNN. Por último, hemos dedicado un capítulo al presente que nos acontece, los Transformers y sus derivados.

Para ir poniendo punto final a este proyecto, vamos a ir enumerando las preguntas que dan sentido a este proyecto:

- ¿Quién rinde mejor modelos clásicos o modelos basados en Deep Learning?

En este punto, dar una respuesta con carácter binario no es adecuada. Decir que unos rinden mejor que otros en la inmensidad del campo que abarcamos no es posible. Rebobinando, las series temporales por naturaleza tienen o no un carácter aleatorio; esta aleatoriedad es la espada de doble de filo que juega en contra de los modelos basados en Deep Learning, ya que como recordamos se generan los denominados “caminos aleatorios” y esto desemboca en una incógnita con respecto al comportamiento de la red.

Esto no significa que los modelos basados en Deep Learning sean en todos los casos inferiores en la predicción de series temporales. De hecho, en escenarios donde el número de datos se dispara, patrones no lineales y relaciones complejas, las arquitecturas como LSTM o CNN demuestran que pueden llegar a superar a los modelos clásicos.

- ¿Los Transformers, con relación a las series temporales, están para quedarse?

Los Transformers están en auge, por ello hemos realizado un estudio de su comportamiento. En donde hemos visto que han revolucionado no solo el procesamiento del lenguaje natural, sino también la predicción de series temporales. Con su atención “Multi Head”, los Transformers tienen la capacidad de ponderar diferentes partes de una serie temporal basándose en su relevancia, proporcionando a menudo una visión más contextual de la serie.

No obstante, para series temporales donde los datos sean escasos o no muy amplios, y las relaciones entre variables tengan un carácter simple, vemos que los Transfomers no pueden competir con los modelos clásicos.

En conclusión, no existe un "santo grial" en la predicción de series temporales. La elección del modelo depende en gran medida del tipo de serie, la cantidad de datos disponibles, los recursos computacionales a mano y el nivel de interpretabilidad deseado.

ENLACES:

CNN teoría: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

Positional Encoding:

https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

Are Transformers Effective for Time Series Forecasting?
https://arxiv.org/abs/2205.13504

Teoría de transfomers:

https://towardsdatascience.com/transformers-141e32e69591

¿Cómo funcionan los Transformers?

https://www.aprendemachinelearning.com/como-funcionan-los-transformers-espanol-nlp-gpt-bert/

Yes, Transformers are Effective for Time Series Forecasting (+ Autoformer)

https://huggingface.co/blog/autoformer#autoformer---under-the-hood

Transformers vs Autoformers:

https://itnext.io/autoformer-decomposition-transformers-with-auto-correlation-for-long-term-series-forecasting-8f5a8b115430