Predicción demanda eléctrica española. Implementación de redes neuronales recurrentes en Python.

Miguel Cabezón Manchado
Trabajo de fin de máster en Ingeniería Matemática
Universidad Complutense de Madrid

Directores del trabajo: Elena Rosa Pérez, Federico Liberatore
Tutor en la empresa: Jorge San Martín Corujo
1 Septiembre de 2018

Resumen

Este proyecto de fin de máster está realizado de manera conjunta a la empresa *Innova-tsn*. El objetivo principal de este proyecto es la predicción de la demanda eléctrica nacional mediante el desarrollo de modelos de redes neuronales recurrentes.
Las redes neuronales están tomando gran importancia en temas de inteligencia artificial, como pueden ser modelos de clasificación y predicción. Por ello el trabajo tratará de explicar de manera teórica qué es una red neuronal y cómo funciona.
Una vez tengamos una primera visión de las redes nos centraremos en las recurrentes, las cuales son más sofisticadas por lo que nos permitirán obtener mejores resultados. Dentro de estas redes recurrentes nos centraremos en las redes de Elman, Jordan, LSTM y GRU.
Después de una primera parte teórica nos centramos en el estudio de la serie a tratar. Para ello se hace un análisis descriptivo de la demanda eléctrica nacional previo a la implementación en *Python* de los modelos de redes elegidos. También se estudiará el efecto de algunas variables explicativas en nuestra predicción.
Por último se presentarán los algoritmos realizados y se compararán predicciones realizadas sin tener un modelo, mediante modelos ARIMA/SARIMA y los obtenidos con las redes neuronales con el objetivo de ver si estas redes mejoran o no los modelos que actualmente manejamos. Esta comparación se realizará mediante un *backtesting* durante un año. Además se tratará de predecir también una demanda a nivel horario mediante dos estrategias.

Abstract

This dissertation has been developed in collaboration with *Innova-tsn* consulting. The main goal of the proyect is the prediction of the national electrical demand through the development of recurrent neural network models.

Neural networks are acquiring importance in artificial intelligence, as they can be used for classification and prediction. Because of that, the dissertation initially provides an explanation of what a neural network is and how it works.

Once we have a first view on neural networks, we focus on recurrent neural networks, which are more sofisticated and they allow us to achieve better results. More specifically, within these recurrent networks we study Elman and Jordan networks and LSTM and GRU cells.

After a first theoric part we start the study of the time serie. To do that, we make a descriptive analysis of the national electrical demand prior to the implementation of the network models chosen in *Python*. We also study the effect of some input variables in our prediction.

Finally, we present the developed algorithms and we compare the predictions obtained without a model, with ARIMA/SARIMA models, and those obtained with neural networks, to understand if they actually improve on the results. This comparison is done by backtesting over a year. Moreover, we also predict the hourly national electrical demand with two different strategies and compare their performance.

Índice general

Capítulo 1

Introducción y objetivos

Este trabajo se ha realizado de la mano con la empresa **Innova-tsn**, tomando importancia con el proyecto de un cliente del sector eléctrico cuyo objetivo es la predicción de la demanda eléctrica a nivel nacional. Esta predicción se realiza a un nivel diario y horario con un horizonte de 1 día, es decir, se predice hoy para mañana. Actualmente la predicción se realiza con modelos SARIMA y el objetivo de la empresa es ver si mediante la implementación de redes neuronales es posible mejorar la predicción actual. Consideraremos una mejor predicción aquella con un error absoluto porcentual medio (MAPE) menor. Todos los resultados existentes en este trabajo se han obtenido mediante la implementación de modelos y algoritmos en los lenguajes de **SAS**, **SAS Guide**, **Python** y **RStudio**.
Los objetivos principales de este Trabajo Fin de Máster son:

- Estudio de la demanda eléctrica nacional a nivel diario y horario con el principal objetivo de minimizar el error absoluto porcentual medio de la predicción.
- Introducción teórica a las redes neuronales, ver tipos de clasificaciones, tipos de aprendizaje, sus elementos, etc.
- Estudio teórico de las principales redes neuronales recurrentes actuales, como las redes de Elman y Jordan, LSTM y GRU.
- Análisis de la serie temporal a tratar y estudio de los datos.
- Estudio de outliers: influencia de festivos, variables de calendario, etc.
- Implementación en Python de los modelos escogidos.
- Desarrollar un proceso de *Backtesting* que nos posibilite la comparación de distintos modelos.

Hay que tener en cuenta que todos los objetivos son también aportaciones ya que la parte teórica introduce contenidos nuevos con respecto al máster y el estudio es una aplicación práctica real realizada con datos reales. Este trabajo tratará de asentar y ampliar conocimientos aprendidos en las asignaturas de Técnicas de Predicción y Estadística Aplicada y Minería de Datos.

1.1. Organización del trabajo

La organización de este trabajo está formada por una parte teórica sobre redes neuronales y todas sus características. Se verán clasificaciones de redes neuronales, los elementos que las forman o incluso los tipos de procesos de aprendizaje que existen.
Lo siguiente será realizar un análisis de los datos a tratar previo a la implementación en Python. Se estudiarán posibles transformaciones de los mismos para que el proceso sea mucho más limpio y efectivo. Además con este estudio es más fácil entender cómo influyen las variables en el consumo de energía eléctrica.

El siguiente paso será la implementación de algunos modelos, usaremos el software libre Python por su simpleza y sencillez a la hora de construir redes neuronales. Además para cada modelo tenemos que definir una forma de puntuación que nos ayude a comparar los resultados obtenidos mediante backtesting y así decidir nuestro modelo final.

1.2. ¿Qué son el *Machine Learning* y el *Deep Learning*?

El Machine Learning es la ciencia de programación cuyo objetivo es que los ordenadores puedan aprender de los datos. De una manera más técnica, es el campo de estudio que da a los ordenadores la habilidad de aprender sin haber sido programados. Por ejemplo, el filtro de los correos spam es un programa de aprendizaje automático que puede aprender a reconocer mensajes spam dándole ejemplos de emails que son spams.
Los campos de machine learning y deep learning a veces parecen indistinguibles uno de otro. Pero se diferencian en que el aprendizaje profundo (*Deep Learning*) eleva el aprendizaje a un nivel más detallado y complejo. Se puede decir que el machine learning tiene una vertiente que se denomina deep learning. Las dos tecnologías hacen referencia a sistemas capaces de aprender por sí solos. La diferencia entre ambos está en el método de aprendizaje. El de deep learning es más complejo y también más sofisticado. Es también más autónomo, lo que quiere decir que una vez programado el sistema la intervención del ser humano es mínima.
Los ejemplos que el ordenador usa para aprender se denominan conjunto de entrenamiento (*training set*). También se necesita una métrica del error cometido que se instentará minimizar que se llama función de pérdida (*loss function*).

1.3. ¿Por qué usamos *Machine Learning*?

Imaginemos que queremos realizar un filtro para correos spam de la forma tradicional. Para ello lo primero es fijarnos en qué forma tienen los correos spam típicos. Lo más probable es que haya una serie de palabras o formas del cuerpo del mensaje que se repitan. Una vez visto qué patrones siguen estos correos, debemos programar un algoritmo que etiquete un correo como spam cuando este tenga una estructura aparentemente spam. Por último, deberíamos testear nuestro programa, volviendo a realizar los pasos anteriores. Un dibujo esquemático del enfoque tradicional es el siguiente:

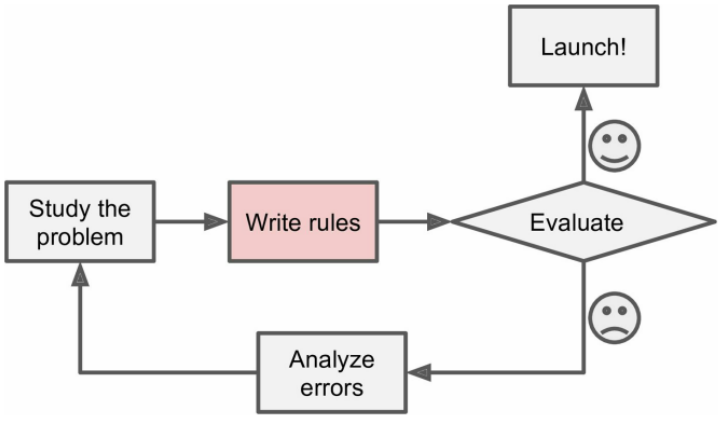


Figura 1.1: Enfoque tradicional. **Fuente:** [2]

Seguramente el problema que tratamos no sea trivial por lo que tendremos que analizar y estudiar muchos errores hasta llegar a tener una larga lista de reglas complejas para conseguir mejorar el modelo. Por otro lado, las técnicas del Machine Learning aprenden automáticamente qué palabras y frases son buenos indicadores de correos spam detectando frecuencia de patrones que son poco comunes en correos

normales y que, sin embargo, se repiten con frecuencia en los spam. Esto lo detecta en los ejemplos de correos spam comparándolos con los que no lo son. Por tanto, el programa es mucho más corto, más fácil de mantener y, posiblemente, más eficaz. El esquema del aprendizaje automático es el siguiente:

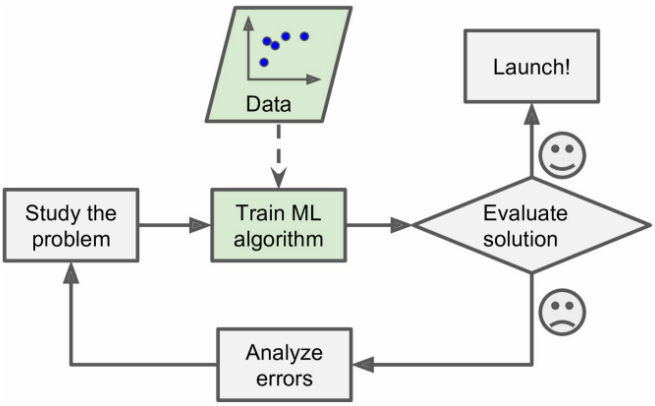


Figura 1.2: Enfoque Machine Learning. **Fuente:** [2]

Además si los que mandan spam se dan cuenta de que tu programa detecta una serie de palabras, lo que harán será cambiar esa serie de palabras detectadas por otras con el mismo significado, lo cual hace que tendríamos que estar revisando y estudiando continuamente cómo se caracteriza un mensaje spam. Sin embargo, con el aprendizaje automático no tenemos que preocuparnos por esto, ya que si las palabras cambian, el ordenador volverá a detectarlas gracias al conjunto de entrenamiento.
Es decir, con el aprendizaje automático podemos adaptarnos al cambio rápidamente:

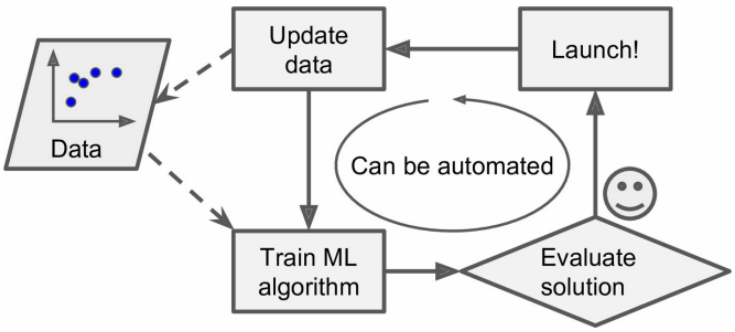


Figura 1.3: Adaptación al cambio de manera automática. **Fuente:** [2]

En resumen, Machine Learning nos ayuda en muchos ámbitos:

- Problemas en los que existe solución pero es muy difícil su mantenimiento o problemas con muchas reglas que dificultan su programación. El Machine Learning simplifica los códigos y obtiene mejores resultados.
- Problemas complejos de los que no existe buena solución. Las nuevas técnicas pueden encontrar una solución.
- Problemas cuyo entorno va cambiando. El sistema de Machine Learning se adapta a los nuevos datos.
- Machine Learning puede recoger información sobre problemas muy complejos y con gran cantidad de datos.

1.4. Tipos de Machine Learning

Hay muchos tipos de aprendizaje automático en cuanto a si el sistema entrena con la supervisión de las personas o sin ella, tenemos cuatro tipos:

- **Aprendizaje Supervisado**:
  Este tipo es el más utilizado y es aquél en el que el conjunto de entrenamiento que le damos a la máquina incluye las soluciones. Estas soluciones se denominan etiquetas. Las redes neuronales que estudiaremos pertenecen a este tipo de entrenamiento. Otros ejemplos de este tipo son: regresión lineal, el método de los k-vecinos más cercanos, regresión logística, árboles de decisión y random forests, etc.

- **Aprendizaje No Supervisado**:
  Es aquél en el que el conjunto de entrenamiento no incluye las soluciones, es decir, está desetiquetado por lo que el sistema intenta aprender sin "profesor". Ejemplos de este tipo son el clustering, PCA (Análisis de Componentes Principales) para reducir la dimensión, etc.

- **Aprendizaje Semisupervisado**:
  Es una mezcla entre los dos anteriores, parte de los datos están etiquetados (con respuesta) mientras que otros no tienen la solución. Algunos ejemplos son redes como Red de Creencia Profunda, en inglés *deep belief networks* (DBN), etc

- **Aprendizaje por Refuerzo**:
  Este es un poco distinto a los anteriores, está inspirado en la psicología conductista y lo que se intenta es determinar qué acciones debe escoger un agente en un entorno dado con el fin de maximizar alguna noción de 'recompensa' (ó minimizar pérdidas). Por lo tanto aprende por sí mismo cuál es la mejor estrategia a seguir, denominado política óptima.

A partir de ahora nos centraremos dentro del Machine Learning en el aprendizaje supervisado de las redes neuronales.

Capítulo 2

Redes Neuronales

La mente humana surge como un modelo para máquinas inteligentes ya que el objetivo que da origen a las redes neuronales es construir un modelo matemático que sea capaz de reproducir el método de aprendizaje del cerebro humano. El resultado de crear un modelo computacional que coincida con la funcionalidad de la mente se denomina neurocomputación.

2.1. El cerebro y las redes neuronales

El cerebro consiste en un conjunto de células cerebrales (**neuronas**) conectadas entre sí. Una neurona toma los impulsos eléctricos como señales, procesan el mensaje y lo envían a las siguientes neuronas. Por tanto una célula cerebral consiste en cuatro partes: las **dendritas** que aceptan los impulsos eléctricos de entrada, el **soma** quien procesa la información, el **axón** que transforma la información de entrada de tal forma que esta información pueda ser recibida por la siguiente neurona, es decir, transforma las inputs en outputs y la **sinápsis** que es el contacto electroquímico entre neuronas. Como vemos en la imagen 2.1.

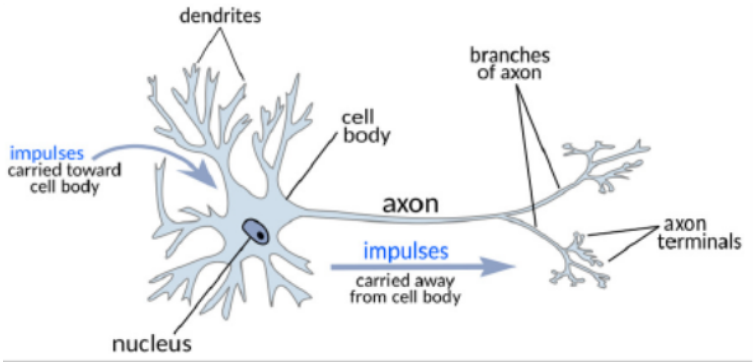


Figura 2.1: Representación de una neurona biológica

Con el objetivo de imitar al cerebro se contruyen las redes neuronales artificiales con una estructura similar aunque más simplificada.
Las **neuronas** artificiales que se interconectan en diversas redes son las encargadas del aprendizaje. Cada neurona recibe y combina señales desde otras neuronas. Mediante la **sinápsis** es posible la transmisión de información entre estas neuronas a través de las **dendritas**. Si la señal combinada es lo suficientemente fuerte, el nervio libera neurotransmisores. Según el tipo de neurotransmisor, las neuronas pueden excitarse si reciben el estímulo o inhibirse si no llega, generando una respuesta u otra según el caso. Vemos la representación de una neurona artificial en la imagen 2.2.

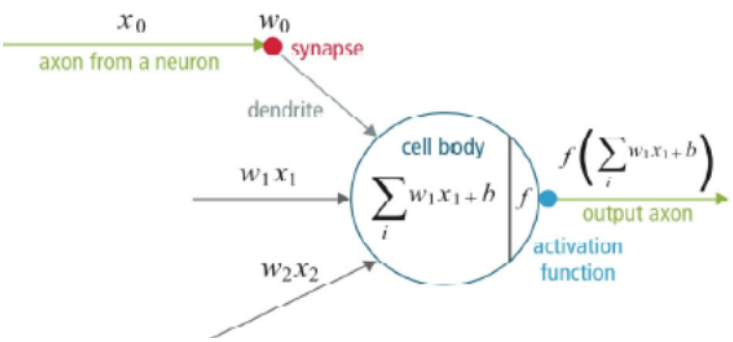


Figura 2.2: Representación de una neurona artificial

2.2. ¿Qué es una red neuronal?

Como hemos visto, una red neuronal está construida por un conjunto de nodos interconectados que se denominan neuronas y se organizan en capas. Una red neuronal básica tiene una capa de entrada (*input layer*), una capa intermedia (*hidden layer*) y una capa de salida (*output layer*). Los nodos de la capa de entrada corresponden con el número de variables que queremos introducir en la red (serían los equivalentes a las variables independientes en una regresión lineal). Por otro lado, el número de nodos de salida depende del número de variables a predecir. Por último la capa intermedia se usa principalmente para aplicar transformaciones no lineales a las variables de entrada originales.
La siguiente figura nos muestra una red neuronal *feedforward* básica para predecir la edad de un niño. Vemos que tiene dos nodos de entrada (que actuarían como las variables altura y peso), una capa oculta con tres nodos y un nodo de salida con la predicción de la edad resultante. Vemos que las conexiones van de principio a fin en una única dirección. El nodo de salida obtiene una suma ponderada por los pesos de las conexiones y de las neuronas para predecir la edad del niño:

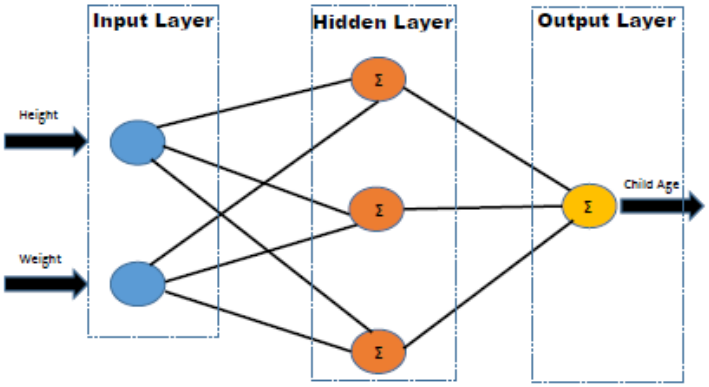


Figura 2.3: Red neuronal feedforward básica

2.2.1. Topología de una red neuronal

La organización y disposición de las neuronas dentro de una red se denomina topología. Como hemos comentado, las redes se estructuran en capas, debiendo tener como mínimo la capa de entrada y la capa de salida que estará asociada a la variable target. Las capas intermedias no son obligatorias y sirven para reflejar relaciones no lineales entre las variables.
En función de en qué capa esté una neurona, esta puede ser de entrada, de salida u ocultas. También

existe la posibilidad de añadir una neurona sesgada que actúe de término constante y haga el papel de *bias* (término constante).
En cuanto a las conexiones pueden ser entre dos neuronas de una misma capa (conexión lateral o intra-capa) o entre neuronas de distintas capas (conexión inter-capa). Según el sentido de estas una red puede ser *feedforward* si todas las conexiones van en un sentido o *feedback* (recurrente) si es posible la conexión en ambos sentidos.

2.2.2. Rol de una neurona

El corazón de una red neuronal artifical es un nodo matemático o neurona. Es el elemento principal de todo el proceso. Las neuronas de la capa de entrada reciben la información, la cual la procesan mediante funciones matemáticas para luego distribuirla a las neuronas de la capa oculta. Esta información se transforma en las capas intermedias hasta finalmente llegar a las neuronas de salida.
La clave es que la información es procesada mediante una función de activación, una vez hecho esto tenemos el resultado ponderado y distribuido en las neuronas de la siguiente capa, por lo que las neuronas se activan a sí mismas con los pesos ponderados. Esto hace que la fuerza con la que dos neuronas están conectadas depende del peso de la información que se envía.

2.2.3. Elementos de una red neuronal

Como acabamos de comentar, la neurona es el centro de la red neuronal, en la siguiente imagen vemos todos los elementos que hacen referencia a ella, y por tanto son necesarios para el aprendizaje de la red:

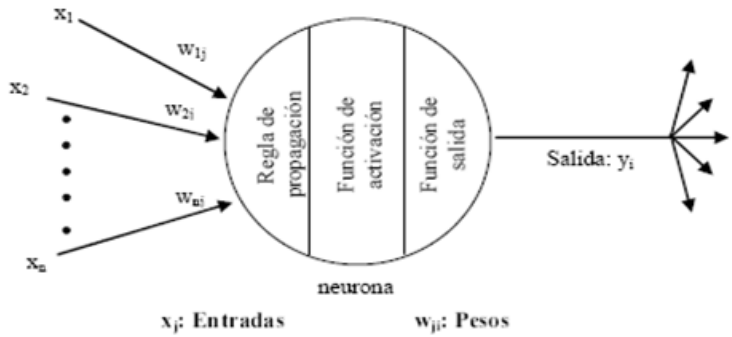


Figura 2.4: Elementos de una neurona artificial

Vemos que las entradas a la red son las variables inputs que nos sirven como variables explicativas. Estas variables tienen que llevar asociados unos pesos iniciales que pueden ser aleatorios o fijados. Por tanto el primer elemento nuevo que nos encontramos son los **pesos sinápticos** w_{ij} que representan la fuerza de una conexión sináptica entre la neurona presináptica i y la postsináptica j. Una neurona puede recibir múltiples entradas simultáneamente y cada entrada tiene su propio peso sináptico el cual proporciona la importancia de la entrada dentro de la función de agregación de la neurona.
Estos pesos representan un estado de la memoria, ya que si un peso es cercano a 0 se considera que no existe conexión (relación) entre esas neuronas. Estos coeficientes pueden adaptarse dentro de la red y van siendo modificados en respuesta de los ejemplos de muestreo de acuerdo a las reglas de entrenamiento.
Una vez explicados los pesos, podemos ver la entrada y salida de las neuronas de forma más detallada siguiendo el ejemplo de la figura 2.5:

- **Una función de entrada**

La neurona trata a los valores de entrada como si fueran uno solo, esto se denomina entrada global. Por lo tanto, ahora queremos combinar estas simples entradas (x_{i1}, x_{i2}, ...) dentro de la entrada global, x_{ij} indica que la variable explicativa x_j entra a la neurona i. Esto se logra a través de la función de entrada, la cual se calcula a partir del vector entrada. La función de entrada puede describirse como sigue:

$$e_i = (x_{i1} \cdot w_{i1}) * (x_{i2} \cdot w_{i2}) * \cdots * (x_{in} \cdot w_{in})$$

donde e_i es la entrada a la neurona i, * representa al operador apropiado, que puede ser la suma, el producto, el máximo, etc, y siendo n el número de entradas a la neurona N_i y w_i al peso.
Como hemos explicado, los pesos, que generalmente no están restringidos cambian la medida de influencia que tienen los valores de entrada. Es decir, que permiten que un gran valor de entrada tenga solamente una pequeña influencia si estos son lo suficientemente pequeños.

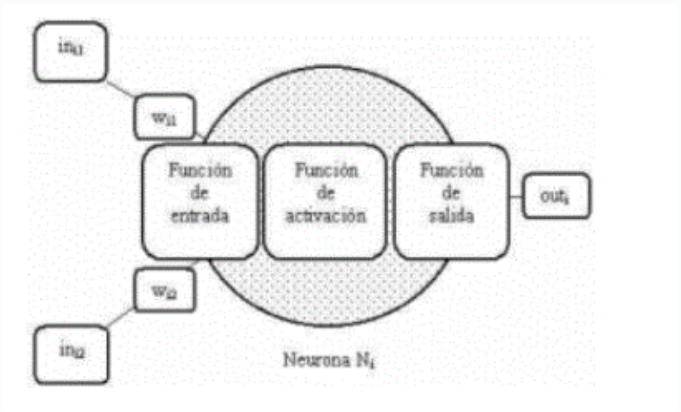


Figura 2.5: Ejemplo de una neurona con 2 entradas y una salida.

- **Una función de activación:**

La función activación f_i calcula el estado de actividad de una neurona. Determina el estado de activación actual $a_i(t)$ de la neurona en base al potencial resultante h_i y al estado de activación anterior de la neurona $a_i(t-1)$. El estado de activación de la neurona i para un determinado instante de tiempo t puede ser expresado de la siguiente manera:

$$a_i(t) = f_i(a_i(t-1), h_i(t))$$

Sin embargo, en la mayoría de los modelos se suele ignorar el estado anterior de la neurona, definiéndose el estado de activación en función del potencial resultante h_i:

$$a_i(t) = f_i(h_i(t)).$$

Hay muchas funciones de activación, las más usuales son:

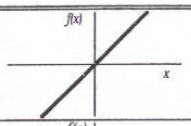
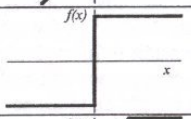
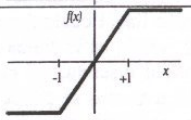
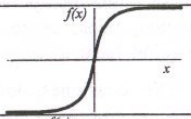
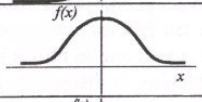
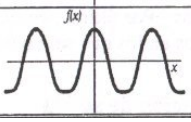
	Función	Rango	Gráfica
Identidad	$y = x$	$[-\infty, +\infty]$	
Escalón	$y = sign(x)$ $y = H(x)$	$\{-1, +1\}$ $\{0, +1\}$	
Lineal a tramos	$y = \begin{cases} -1, & si\ x < -l \\ x, & si\ +l \le x \le -l \\ +1, & si\ x > +l \end{cases}$	$[-1, +1]$	
Sigmoidea	$y = \frac{1}{1+e^{-x}}$ $y = tgh(x)$	$[0, +1]$ $[-1, +1]$	
Gaussiana	$y = Ae^{-Bx^2}$	$[0,+1]$	
Sinusoidal	$y = A\,\mathrm{sen}(\omega x + \varphi)$	$[-1,+1]$	

Figura 2.6: Funciones de activación.

- **Una función de salida**

  El último componente que una neurona necesita es la función de salida. El valor resultante de esta función es la salida de la neurona i (out). Por tanto, la función de salida determina qué valor se transfiere a las neuronas vinculadas. Si la función de activación está por debajo de un umbral determinado, ninguna salida se pasa a la neurona siguiente. Normalmente, los valores de salida están comprendidos en el rango $[0,1]$ o $[-1,1]$ o también pueden ser binarios $\{0,1\}$ o $\{-1,1\}$. Dos de las funciones de salida más comunes son:

  - Ninguna: este es el tipo de función más sencillo, tal que la salida es la misma que la entrada. Corresponde con la función identidad (F≡ Identidad). Quedaría:

    $$prediccion_i(t) = F_i(a_i(t)) = a_i(t)$$

  - Binaria:

    $$F_i(a_i(t)) = \begin{cases} 1 & \text{si } a_i(t) \ge \varepsilon_i, \\ 0 & \text{en caso contrario.} \end{cases}$$

    donde ε_i es el umbral de aceptación.

Hemos visto los elementos que conforman una red neuronal, pero para el entrenamiento de la red, se requieren tres nuevos elementos:

- **Función de coste** (*cost function*): La función de coste nos dirá cómo de malo es nuestro modelo en términos de su habilidad para estimar la relación entre las variables de entrada x y el target y. El objetivo de la red será minimizar esta función. Algunos ejemplos son *MSE* (mean squared error), *MAE* (mean absolute error), *MAPE* (mean absolute percentage error) para variables continuas, para binarias existen *binary cross entropy*, *categorical crossentropy*, etc
- **Optimizador** (*optimizer*): Es el mecanismo a través del cual la red se actualiza, basándose en los datos que ve y en la función de coste. Un ejemplo es el gradiente descendiente.

- **Métrica** (*metric*): Como su nombre indica, es una medida para puntuar cómo de bueno es un modelo. Nos sirve para comparar distintos modelos y poder decantarnos por el mejor. Se suele tener en cuenta la precisión (*accuracy*).

2.3. Aprendizaje de una red neuronal

Las redes neuronales manejan dos tipos de información. La primera, es la información volátil, que se refiere a los datos que se están usando y varían con la dinámica de la red. Esta información se encuentra almacenada en el estado dinámico de las neuronas. El segundo tipo es la información no volátil que se mantiene para recordar los patrones aprendidos y se encuentra almacenada en los pesos sinápticos.

El proceso de aprendizaje o entrenamiento de una red neuronal consiste en encontrar la relación entre los pesos que permiten desarrollar la tarea para que la red ha sido diseñada, como puede ser predecir o clasificar.

El aprendizaje se basa en el entrenamiento de la red con patrones, que usualmente son llamados patrones de muestra o entrenamiento. El proceso usual del algoritmo es que la red ejecuta los patrones iterativamente, cambiando los pesos de las sinapsis, hasta que convergen a un conjunto de pesos óptimos que representan a los patrones lo suficientemente bien. Esto es, sus pesos sinápticos se ajustan para dar respuestas correctas al conjunto de patrones de entrenamiento que le hemos mostrado.

2.3.1. Etapas del aprendizaje y método de entrenamiento

Para este aprendizaje son necesarias técnicas como minimización del error o la optimización de alguna "función de recompensa"para modificar el valor de los pesos sinápticos en función de las entradas disponibles y con ello optimizar la respuesta de la red a las salidas que deseamos.

Por ello necesitamos establecer una función de error $E(W)$ que mida el rendimiento de la red en un instante dado. Lógicamente, el objetivo del proceso de entrenamiento es encontrar los pesos que minimicen la función de error escogida. Las etapas en este proceso de aprendizaje son:

- **1. Inicialización de la red:** Al comienzo del aprendizaje, en $n = 0$ se generan unos pesos aleatorios para todas las conexiones. Con estos pesos se puede calcular un primer error de predicción o clasificación.

- **2. Hacia delante:** La información pasa desde la entrada hasta la salida en una única dirección a través de las funciones de activación y de los pesos. Para cada instante de tiempo n se analiza un nuevo patrón de entrada y se intentan refinar los pesos buscando mejorar el nivel de error de la etapa anterior.

- **3. Asignación del error:** El algoritmo se detiene cuando el error obtenido es menor que una cota prestablecida, cuando el error no decrece sensiblemente o cuando se ha alcanzado el número de iteraciones. En caso contrario se pasa al paso 4.

- **4. Propagación del error:** El error obtenido en la capa de salida es propagado hacia atrás para modificar los pesos. El algoritmo más utilizado es el *Backpropagation* que explicaremos a continuación.

- **5. Ajuste:** Se ajustan los pesos usando el método del gradiente descendiente buscando la minimización del error.

Una vez visto el proceso general del aprendizaje vamos a ver con más detalle el método de entrenamiento más usado, que es el algoritmo backpropagation.
Este tiene dos etapas, primero calcula la salida de la red dado un patrón de entrada y segundo calcula su error (entre la salida obtenida y el target) y propaga este error hacia atrás desde la capa de salida, donde cada neurona precedente recibe un error proporcional a su contribución sobre el error total de la red.

Este método está basado en el método del gradiente descendente el cual se basa en que si una función multivariable $F(x)$ está definida y es diferenciable en un entorno de a, entonces $F(x)$ disminuye más rápidamente en la dirección del gradiente negativo de F en a. De aquí tenemos que :

$$a_{n+1} \leq a_n - \mu \nabla F(a_n)$$

para μ suficientemente pequeño, por lo que $F(a_n) \geq F(a_{n+1})$. En otras palabras, restamos el término $\mu \nabla F(a_n)$ a a_n porque queremos movernos en dirección contraria al gradiente, buscando el mínimo.

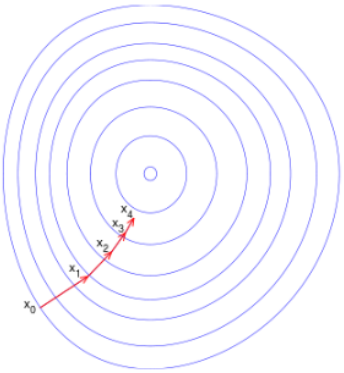


Figura 2.7: Ejemplo del método del gradiente descendente.

En nuestro caso se parte de un conjunto de pesos $W(0)$ en el instante inicial y se calcula la dirección de máximo decrecimiento del error, que es su vector gradiente $\nabla E(W)$. Por lo comentado del método del gradiente se deben actualizar los pesos siguiendo el sentido contrario al gradiente y obtendríamos lo siguiente:

$$W(n+1) \leq W(n) - \mu \nabla E(W(n)).$$

El objetivo es descender por la superficie del error hasta alcanzar un mínimo local. De manera esquemática podemos decir que el método del gradiente descendente consta de tres pasos:

- Determinar el valor del paso μ.
- Calcular el gradiente de la función error para encontrar la dirección de máximo decreciemiento.
- Moverse en la dirección correspondiente hasta que el vector gradiente sea prácticamente nulo

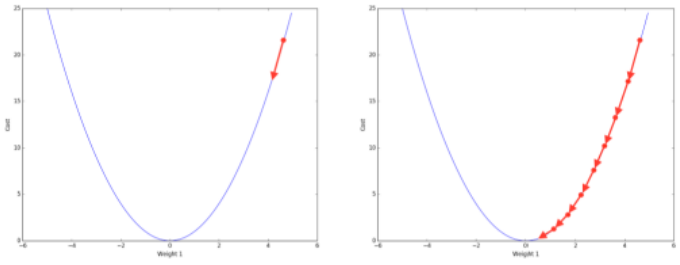


Figura 2.8: Ejemplo 2 del método del gradiente descendente.

μ es la tasa de aprendizaje (*learning rate*) e indica el tamaño del paso. Este parámetro mide la velocidad con que varían los pesos y tiene bastante importancia ya que si μ es muy pequeño la velocidad de aprendizaje es lenta y puede caer en mínimos locales, mientras que si μ es grande hay una mayor variación de los pasos en cada iteración y es más rápido pero puede no encontrar nunca una solución si oscila en torno al mínimo.

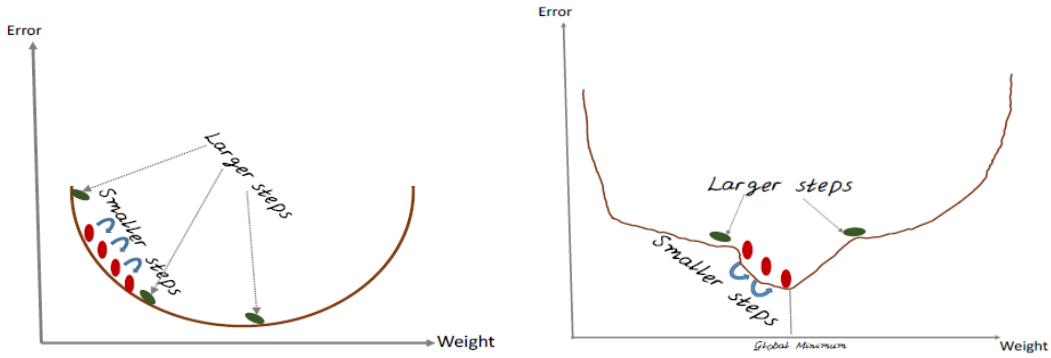


Figura 2.9: Problemas de la tasa de aprendizaje. **Fuente:** [5]

Como vemos no hay un valor fijo para la tasa de aprendizaje ya que depende de la muestra que se use para entrenar la red.
Por último vamos a hablar del *momentum* o aceleración del aprendizaje.
La idea de este término es que la variación de los pesos no sea tan brusca y poder reducir el impacto de las oscilaciones para tasas de aprendizaje altas. Gracias a este término se genera un cambio en la dirección para que el cambio actual sea más suave. El momentum determina el efecto en el instante $t+1$ del cambio de los pesos realizados en el instante t. Este término garantiza la convergencia de la red en un número menor de iteraciones aunque son más lentas. Con momentum m, la actualización del peso en tiempo t es:

$$\Delta W_{ij}(t+1) = \mu_i \frac{\partial E(t)}{\partial h_j} x_i(t) + m\Delta W_{ij}(t).$$

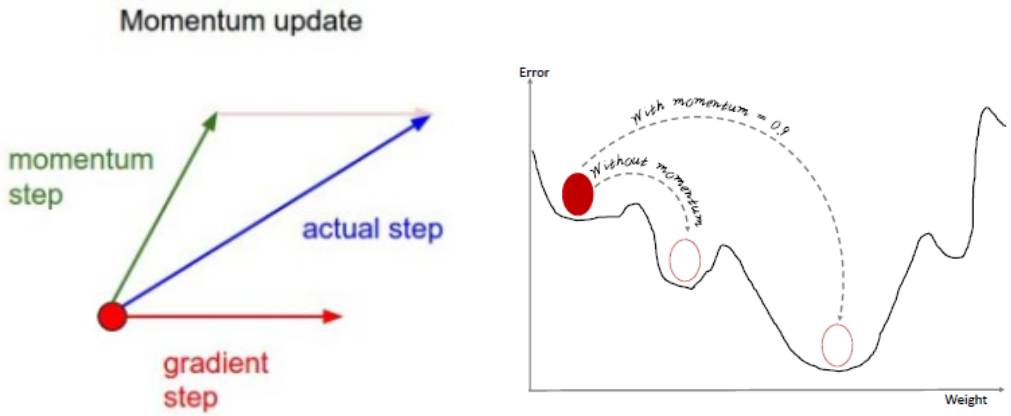


Figura 2.10: Influencia del momentum y ejemplo. **Fuente:** [5]

2.4. Tipos de redes neuronales

Las redes neuronales se pueden clasificar de muchas maneras, una clasificación es según su **topología**, es decir podemos distinguir como característica de una red el número de capas, el tipo de capas, que pueden ser ocultas o visibles, de entrada o de salida y la direccionalidad de las conexiones de las neuronas. Otra posible clasificación es según su algoritmo de **aprendizaje**, o como la red aprende los patrones, esta clasificación de las redes es muy parecida a la que vimos de Machine Learning.

2.4.1. Clasificación respecto a la topología

La arquitectura de una red consiste en la organización y disposición de las neuronas en la red. Las neuronas se agrupan formando capas, que pueden tener distintas características. Además las capas se organizan hasta formar una red. En la siguiente imagen vemos la jerarquía de las redes neuronales:

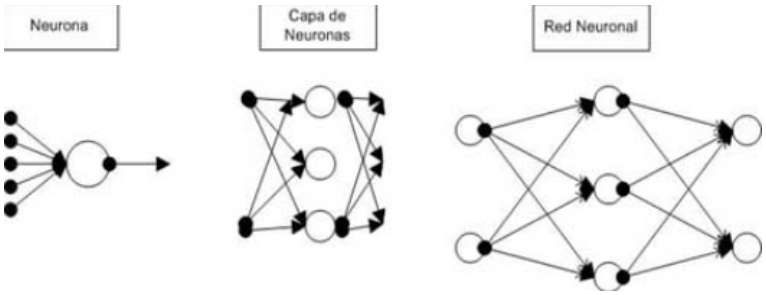


Figura 2.11: Jerarquía de las redes neuronales

Podemos ver que las neuronas se agrupan para formar capas y que estas se unen formando redes neuronales. De este modo usando la topología de la red clasificamos en:

- Redes Monocapa: Son aquellas con una sola capa. Para unirse se crean conexiones entre las neuronas de la capa. Entre las redes monocapa existen algunas en las que las neuronas pueden estar conectadas consigo mismas y se denominan autorecurrentes.
- Redes Multicapa: Las redes multicapa están formadas por varias capas de neuronas. Estas redes se pueden a su vez clasificar atendiendo a la manera en que se conectan sus capas.
  - *Feedforward*: Cuando las conexiones entre las capas siguen un único sentido (hacia delante), desde los inputs hasta los outputs. Ejemplos de este tipo de red son el perceptrón, adaline, etc. También se denominan estáticas.
  - *Feedback* o *recurrente*: Cuando las conexiones pueden ser tanto hacia delante como hacia atrás y por tanto la información puede volver a capas anteriores en la dinámica de la red. Se pueden denominar también como dinámicas.

2.4.2. Clasificación respecto al aprendizaje

Como vimos en los tipos de aprendizaje que tienen las técnicas de machine learning, en las redes neuronales también podemos distinguir tres tipos de aprendizaje.
El modo más intuitivo es el aprendizaje supervisado, que consiste en que la red dispone de los patrones de entrada y los patrones de salida que deseamos para esa entrada y en función de ellos se modifican los pesos de las sinapsis para ajustar la entrada a esa salida. Este aprendizaje es con el que vamos a tratar en este trabajo, en concreto el aprendizaje por corrección de error.
Otro modo de aprendizaje, aprendizaje no supervisado, consiste en no presentar patrones objetivos, sino sólo patrones de entrada, y dejar a la red clasificar dichos patrones en función de las características comunes de los patrones. Dentro de este tipo está el aprendizaje por componentes principales y el aprendizaje competitivo.
Por último existe otro tipo de aprendizaje, que usa una fórmula híbrida, el supervisor no enseña patrones objetivos si no que sólo dice si acierta o falla en su respuesta ante un patrón de entrada. Es el aprendizaje reforzado.

Capítulo 3

Redes Neuronales Recurrentes

Las redes neuronales recurrentes (Recurrent Neural Networks) tienen una gran habilidad para aprender problemas difíciles que tratan sobre datos con series temporales. Estas redes tienen estados ocultos distribuidos en el tiempo que permiten almacenar mucha información sobre el pasado de forma eficiente.
Las redes recurrentes se han usado en problemas de control adaptativo, sistemas de identificación y en reconocimiento de voz. También son útiles a la hora de traducción de textos y de clasificación de sentimientos, como por ejemplo, puntuar una película por el comentario que haya dejado un usuario. Además su habilidad de anticipación les da la capacidad de una creatividad sorprendente. Por ejemplo, podemos predecir las siguientes notas de una melodía una y otra vez hasta que una red componga su propia melodía como la producida por el proyecto Magenta de Google (`https://magenta.tensorflow.org/`).

3.1. ¿Qué es una red neuronal recurrente?

Como hemos visto, las redes neuronales pueden ser dinámicas o estáticas. A diferencia de las redes que vimos en el capítulo anterior (estáticas), las redes recurrentes son dinámicas, es decir también tienen conexiones hacia atrás. La salida depende de los actuales valores de las variables de entrada a la red, de los anteriores valores de la entrada, de los anteriores valores de la salida, etc.
Fijémonos en la más sencilla red neuronal recurrente posible, compuesta por sólo una neurona que recibe los inputs y que produce un output y que este lo envía así misma como vemos en la parte izquierda de la siguiente imagen:

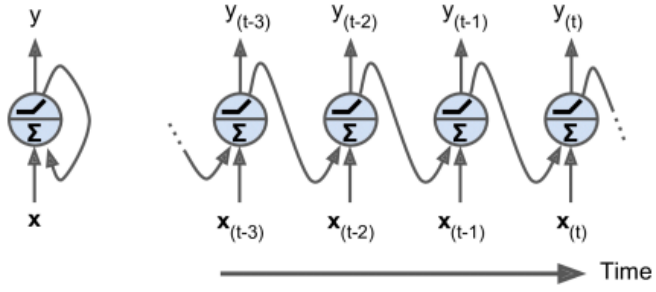


Figura 3.1: Neurona recurrente (izquierda), desarollo en el tiempo (derecha). **Fuente:** [2]

Vemos que en cada paso temporal t (también llamado marco temporal), esta neurona recurrente recibe tanto la entrada $x_{(t)}$ como la salida del paso anterior, $y_{(t-1)}$. Si representamos este proceso en el eje x como si fuera el tiempo, obtendríamos la parte derecha de la imagen 3.1.
Con esto podemos formar fácilmente una capa con neuronas recurrentes. En cada tiempo t todas las neuronas reciben el vector de entrada $x_{(t)}$ y el vector de salida del paso anterior $y_{(t-1)}$ como se muestra

a continuación:

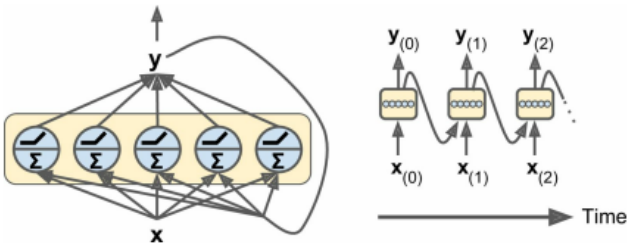


Figura 3.2: Capa de neuronas recurrentes (izquierda), desarollo en el tiempo (derecha). **Fuente:** [2]

Cada neurona recurrente tiene dos conjuntos de pesos; uno para los inputs $x_{(t)}$ y otro para las salidas del paso temporal anterior $y_{(t-1)}$. Llamemos a estos pesos w_x y w_y. La salida $y_{(t)}$ de una neurona recurrente se puede calcular de la siguiente manera:

$$y_{(t)} = \phi(x_{(t)}^T \cdot w_x + y_{(t-1)}^T \cdot w_y + b)$$

donde b es el sesgo (bias) y $\phi(\cdot)$ es la función de activación. Podemos resumir la expresión anterior mediante el álgebra lineal de la siguiente manera:

$$Y_{(t)} = \phi(X_{(t)} \cdot W_x + Y_{(t-1)} \cdot W_y + b) = \phi([X_{(t)} \quad Y_{(t-1)}] \cdot W + b)$$

siendo $W = [W_x \quad W_y]^T$ y donde :

- $Y_{(t)}$ es una matriz m x $n_{neuronas}$ que contiene los outputs del marco temporal t para cada batch (que es el conjunto de datos que entran a la neurona) en este caso m y $n_{neuronas}$ es el número de neuronas de la capa.
- $X_{(t)}$ es una matriz m x n_{inputs} que contiene las variables inputs de las m observaciones y n_{inputs} es el número de variables explicativas de entrada.
- W_x es una matriz n_{inputs} x $n_{neuronas}$ conteniendo los pesos de las conexiones para las entradas del paso temporal actual.
- W_y es una matriz $n_{neuronas}$ x $n_{neuronas}$ conteniendo los pesos de las conexiones para las salidas del paso temporal anterior.
- Las matrices de pesos W_x y W_y se pueden juntar en la matriz W que será $(n_{inputs} + n_{neuronas})$ x $n_{neuronas}$
- b es un vector de 1's de tamaño $n_{neuronas}$ que permite añadir un término constante (bias) al modelo.

Hay que darse cuenta de que $Y_{(t)}$ es una función de $X_{(t)}$ y de $Y_{(t-1)}$, quien a su vez es función de $X_{(t-1)}$ y de $Y_{(t-2)}$, quién es función de $X_{(t-2)}$ y de $Y_{(t-3)}$, etc. Esto hace que $Y_{(t)}$ sea función de todas las variables de entrada desde el tiempo inicial $t = 0$ (esto es, de $X_{(0)}, X_{(1)}, \ldots, X_{(t)}$). En $t = 0$ no hay outputs anteriores luego se asumen como ceros.
Una vez visto como funciona una red recurrente de manera genérica, nos centramos en el estudio de algunas de las redes neuronales recurrentes más conocidas.

3.2. Redes Neuronales Recurrentes de Elman y Jordan

A continuación se presentan dos tipologías de redes neuronales recurrentes sencillas.

3.2.1. Red de Elman

Una red neuronal de Elman es una red con tres capas (entrada, oculta y salida) a las que se le añade una capa con 'unidades de contexto' (o también llamadas unidades de retardo). Cada capa contiene una o más neuronas que propagan la informarción de una capa a la siguiente con la regla de aprendizaje que se haya fijado.
En las redes de Elman el número de neuronas de la capa de contexto tiene que ser igual al número de neuronas de la capa oculta. Además, todas las neuronas de la capa de contexto están conectadas con una neurona de la capa oculta.
El proceso de memoria se produce a través de las unidades de retardo que son alimentadas por las neuronas de la capa oculta. Los pesos de las conexiones entre la capa oculta y las unidades de retardo son fijos e iguales a 1. Gracias a esto en las unidades de retardo siempre mantenemos una copia de los valores de salida de las neuronas de la capa oculta del paso previo.
Podemos ver una representación de la red neuronal de Elman en la imagen 3.3:

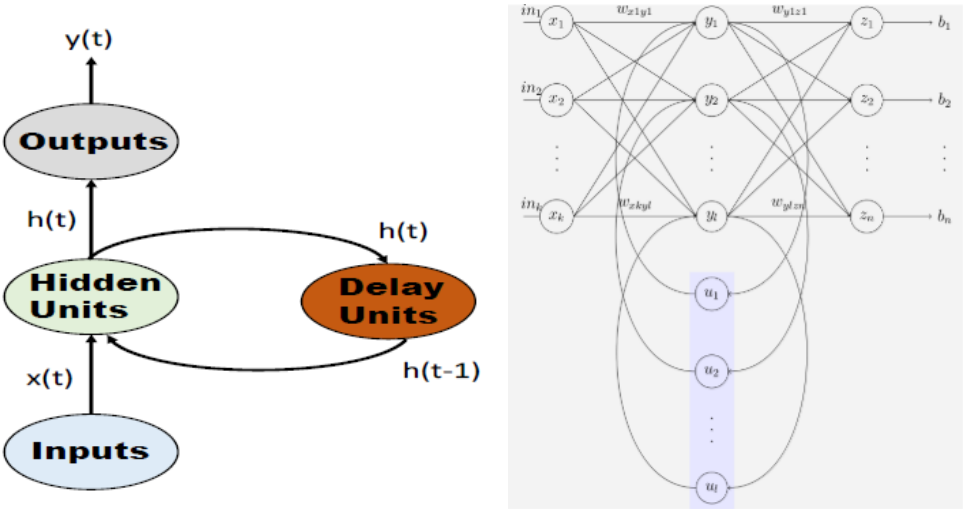


Figura 3.3: Dos representaciones de la red neuronal de Elman. **Fuente:** [5]

En la imagen de la izquierda vemos el esquema de la red de Elman más generalizado, observando que la salida de la capa oculta se almacena en la capa de retardo, y que luego esta sirve de entrada en el siguiente paso para la capa oculta. En la imagen de la derecha podemos ver las unidades de retardo u_i en las que se almacena el valor de la salida de la neurona y_i y sirven como entrada en el siguiente paso.

3.2.2. Red de Jordan

La red neuronal de Jordan tiene una arquitectura similar a la de Elman, es decir, consta de la capa de entrada, una capa oculta, la capa de salida y la capa de contexto. La única diferencia es que ahora se guarda la información de la capa de salida en vez de la información de la capa oculta, por lo que la capa de contexto está conectada a la capa oculta y a la de salida como vemos en la imagen 3.4. Por tanto, esta red recuerda la salida final del paso anterior en vez de la salida de la capa oculta. La estructura de la red neuronal de Jordan es la siguiente:

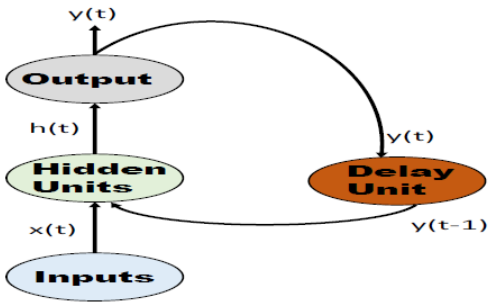


Figura 3.4: Estructura de la red neuronal de Jordan. **Fuente:** [5]

Las ecuaciones matemáticas que definen ambos procesos son:

- **Red de Elman**:

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h),$$
$$y_t = \sigma_y(W_y h_t + b_y).$$

- **Red de Jordan**:

$$h_t = \sigma_h(W_h x_t + U_h y_{t-1} + b_h),$$
$$y_t = \sigma_y(W_y h_t + b_y).$$

donde x_t es el vector con la variables de entrada, h_t es el vector de la capa oculta, y_t es el vector de salida, $W = [W_x \quad W_y]^T$ y U_h son las matrices de pesos y $b = [b_h \quad b_y]^T$ el bias. σ_h y σ_y son las funciones de activación de las respectivas capas.
A las redes de Elman y Jordan también se les conoce como redes neuronales recurrentes simples. Los siguientes apartados tratan dos tipos de redes recurrentes con una técnica de memoria más compleja.

3.3. Células LSTM

Una red neuronal recurrente *Long Short Term Memory* es parecida a las redes de Elman y de Jordan sólo que sustituye las neuronas y las unidades ocultas por un bloque de memoria, este bloque es una célula LSTM. En la siguiente imagen vemos cómo es una célula LSTM por dentro:

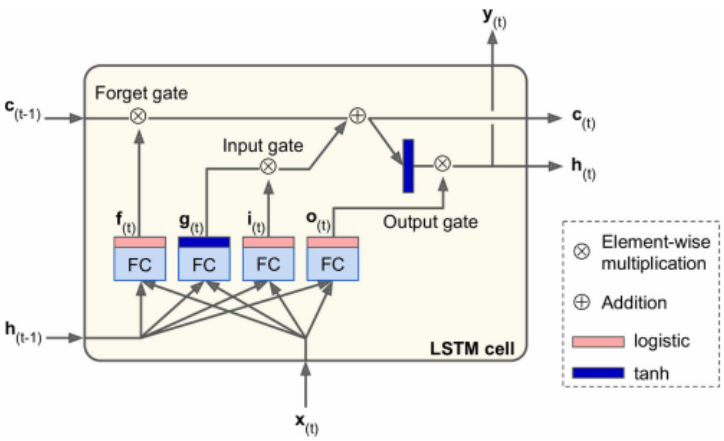


Figura 3.5: Célula LSTM. **Fuente:** [2]

Si no nos fijamos en el interior de la caja, una célula LSTM es igual a una célula normal salvo que su estado ahora se divide en dos vectores, $h_{(t)}$ y $c_{(t)}$ (c viene de 'célula'). $h_{(t)}$ hace referencia al estado de corto plazo y $c_{(t)}$ al de largo plazo.
La principal idea de la célula es que la red pueda aprender qué almacenar en el estado de largo plazo, qué desechar y qué leer de ellos. Si nos fijamos en el término de largo plazo $c_{(t-1)}$ atraviesa la red de izquierda a derecha, primero pasa por una puerta de olvido (*forget gate*), deshaciéndose de algunos datos almacenados, luego añade nuevas informaciones que llegan de la puerta de entrada (*input gate*) y el resultado $c_{(t)}$ se envía directamente fuera de la célula y se deja como está y servirá de entrada para la siguiente época. Por lo tanto, en cada paso algunos recuerdos son eliminados mientras que otros nuevos se añaden. Además, una vez añadidos nuevos recuerdos, se hace una copia del estado a largo plazo y pasa por la función tangente hiperbólica para después ser filtrado por la puerta de salida (*output gate*). Este paso produce el estado de corto plazo $h_{(t)}$ (que es igual a la salida de la célula ($y_{(t)}$) para este paso temporal t).
Ahora vamos a ver de dónde vienen los nuevos recuerdos que se memorizan y cómo trabajan estas puertas.
Primero vemos que el vector de entrada actual $x_{(t)}$ y el estado de corto plazo anterior $h_{(t-1)}$ van a parar a cuatro capas conectadas. Todas ellas con un distinto objetivo:

- La capa principal es la que tiene como salida a $g_{(t)}$. Tiene el típico rol de analizar las entradas actuales $x_{(t)}$ y el estado a corto plazo anterior $h_{(t-1)}$. En una célula normal sólo habría esta capa y su salida iría directamente a $y_{(t)}$ y $h_{(t)}$, sin embargo en una célula LSTM la salida de esta capa no va directamente fuera, se almacena parcialmente el estado a largo plazo.

- Las otras tres capas son controladores de puertas (*gate controllers*) ya que usan la función logística como función de activación luego su salida varía entre 0 y 1. Como vemos en la imagen 3.5 sus salidas van directas a operaciones multiplicativas, por lo que si la salida es cercana a 0 menos información pasa 'y se cierra la puerta' mientras que si se acerca a 1 más información fluye 'y se abre la puerta'. Más específicamente:

  - La *forget gate* ($f_{(t)}$) controla qué partes del estado a largo plazo deben ser borradas.
  - La *input gate* ($i_{(t)}$) controla qué partes de $g_{(t)}$ deben ser añadidas al estado de largo plazo (por esto se dijo que se almacenan parcialmente).
  - La *output gate* ($o_{(t)}$) controla qué partes del estado de largo plazo deben ser leídas y su salida es tanto $h_{(t)}$ como $y_{(t)}$.

En resumen, una célula LSTM puede aprender a reconocer datos inputs importantes (que es el rol de la puerta de entrada), almacenarla en el estado de largo plazo, aprender a guardarla siempre que sea necesaria (cuyo rol es el de la puerta de olvido) y aprender a extraer su información siempre y cuando se necesite. Todo esto es la razón de por qué LSTM ha tenido tanto éxito a la hora de encontrar patrones a largo plazo en series temporales, largos textos, grabaciones de audio, etc.
Las siguientes ecuaciones resumen cómo calcular el estado de largo plazo, el de corto plazo y su salida para un paso temporal.

$$
\begin{aligned}
i_{(t)} &= \sigma(W_{xi}^T \cdot x_{(t)} + W_{hi}^T \cdot h_{(t-1)} + b_i) \\
f_{(t)} &= \sigma(W_{xf}^T \cdot x_{(t)} + W_{hf}^T \cdot h_{(t-1)} + b_f) \\
o_{(t)} &= \sigma(W_{xo}^T \cdot x_{(t)} + W_{ho}^T \cdot h_{(t-1)} + b_o) \\
g_{(t)} &= tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot h_{(t-1)} + b_g) \\
c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\
y_{(t)} &= h_{(t)} = o_{(t)} \otimes tanh(c_{(t)}).
\end{aligned}
$$

Donde,

- $W_{xi}, W_{xf}, W_{xo}, W_{xg}$ son las matrices de pesos de las conexiones entre las cuatro capas y el vector de entrada $x_{(t)}$.
- $W_{hi}, W_{hf}, W_{ho}, W_{hg}$ son las matrices de pesos de las conexiones entre las cuatro capas y el estado de corto plazo previo $h_{(t-1)}$.
- b_i, b_f, b_o y b_g son los términos de bias para las cuatro capas.

3.4. Células GRU

Las células GRU (*Gated Recurrent Unit*, unidades recurrentes cerradas) son una versión simplificada de las células LSTM. En la siguiente imagen podemos ver su esquema:

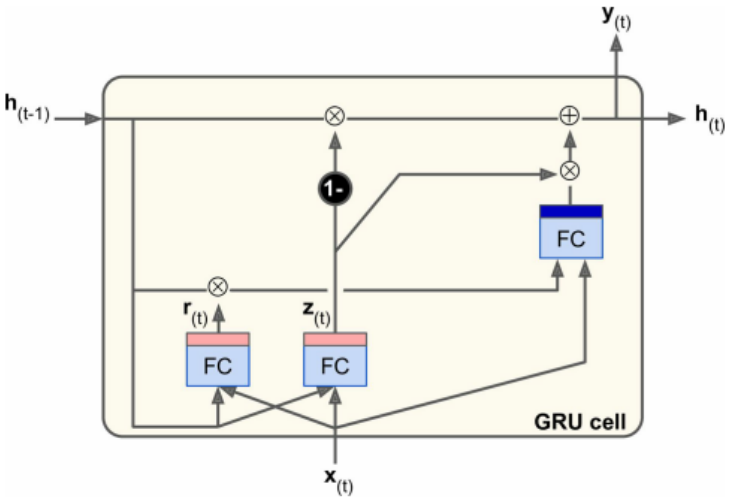


Figura 3.6: Célula GRU. **Fuente:** [2]

Las principales simplificaciones son las siguientes:

- Los dos vectores de estado (corto y largo plazo) se juntan ahora en un único vector $h_{(t)}$.
- Ahora sólo hay un controlador de las puertas que controla tanto la puerta de olvido como la puerta de entrada. Si la puerta de control vale 1, la puerta de entrada se abre y la de olvido se cierra y si la de control vale 0 ocurre lo contrario.
- No hay puerta de salida, la salida siempre es el vector de los estados entero. Sin embargo hay una nueva puerta de control que controla qué partes del estado anterior deben ser mostradas en la capa principal.

Las siguientes ecuaciones resumen lo anterior explicado:

$$
\begin{aligned}
z_t &= \sigma(W_{xz}^T \cdot x_{(t)} + W_{hz}^T \cdot h_{(t-1)}) \\
r_t &= \sigma(W_{xr}^T \cdot x_{(t)} + W_{hr}^T \cdot h_{(t-1)}) \\
g_t &= tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot (r_{(t)} \otimes h_{(t-1)})) \\
h_t &= (1 - z_{(t)}) \otimes tanh(W_{xg}^T \cdot h_{(t-1)} + z_{(t)} \otimes g_{(t)})
\end{aligned}
$$

Las células LSTM y GRU son una de las principales razones detrás del éxito de las redes neuronales recurrentes en los útlimos años.

Capítulo 4

Predicción de la demanda en Python

Una vez visto qué es una red neuronal, algunos ejemplos de ellas y cómo funcionan por dentro, procedemos a empezar a tratar con la parte experimental del trabajo, la cual se concentra en implementar algunas de las redes neuronales vistas y valorar los resultados obtenidos de la predicción a nivel diaria. Una vez tenemos esta predicción, se intentará mejorar el resultado mediante la adición de nuevas variables explicativas. Por otro lado, se probarán dos formas de predicción horaria y veremos cual obtiene mejor resultado.

4.1. Análisis de la serie y estudio de los datos

Antes de empezar a construir redes hay que hacer un estudio general de los datos que tenemos y de la variable a predecir para poder entender posibles variables de entrada que puedan ser útiles en la predicción.

4.1.1. Análisis descriptivo de la serie

Lo primero que hacemos es dibujar nuestra variable a predecir para ver su comportamiento, por tanto en la siguiente imagen se muestra la demanda eléctrica nacional a nivel diario desde el 1 de enero de 2012 hasta el 31 julio de 2018 que es hasta donde tenemos información válida de la demanda que se ha consumido en España.

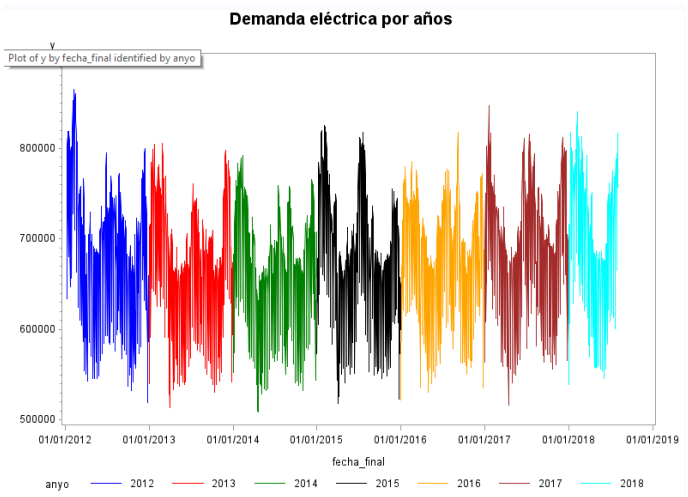


Figura 4.1: Demanda eléctrica nacional en años consecutivos

Se ve claramente que nuestra serie sigue un patrón muy parecido año tras año, lo cual indica que existe

una estacionalidad anual. Si nos centramos en ver cómo es el comportamiento dentro de un año concreto, nos encontramos con que hay algunos meses en los que se consume más que en otros, estos meses son enero, febrero, julio, agosto y diciembre. Si nos fijamos el consumo de electricidad está muy relacionado con dos aspectos, uno es el consumo energético de todas las viviendas residenciales de nuestro país mientras que el otro es toda la actividad industrial que se realiza. Por tanto, debido al consumo que se realiza, los meses de invierno y verano tienen una mayor demanda debido a su relación con la temperatura, ya que en invierno hace mucho más frío y es necesaria la calefacción en casa, además de haber más oscuridad y también requerir más luz en las mismas. En el lado contrario, en verano hace mucho calor y por tanto no es necesaria la calefacción, sin embargo, este calor hace que la mayoría de aires acondicionados estén funcionando, por lo que este consumo eléctrico vuelve a verse incrementado por temas de temperatura. En la imagen siguiente podemos ver el consumo durante el año 2013.

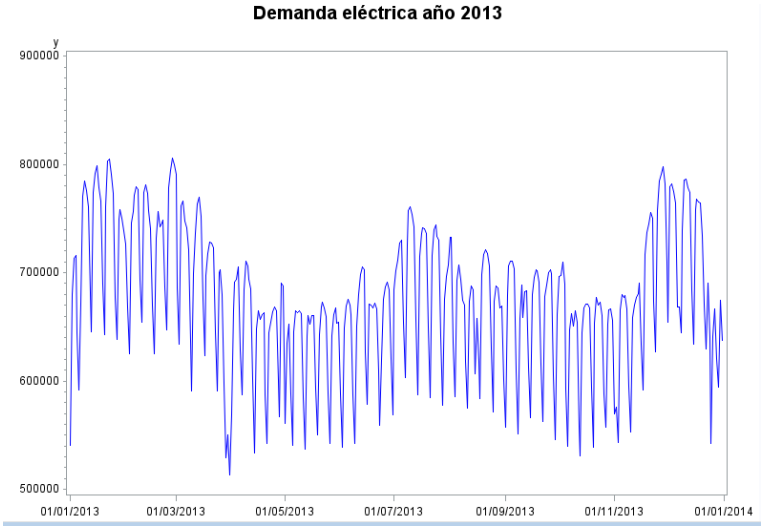


Figura 4.2: Demanda eléctrica nacional en el año 2013

Como se ha comentado, en la demanda eléctrica hay dos factores principales que influyen, la temperatura y la actividad industrial. Esta actividad no suele ser influida por la temperatura, por lo que todo los meses del año se consume más o menos la misma energía eléctrica debida a la industria. Y digo más o menos porque es lógico que en verano este consumo baje un poco debido a periodos vacacionales. Sin embargo, lo que más influye en la demanda eléctrica industrial son el tipo de día ya que cuando es un festivo se para la industria y se produce un bajón de demanda eléctrica. Para ver esto más claro dibujaremos la demanda de los meses de abril y mayo del año 2013.

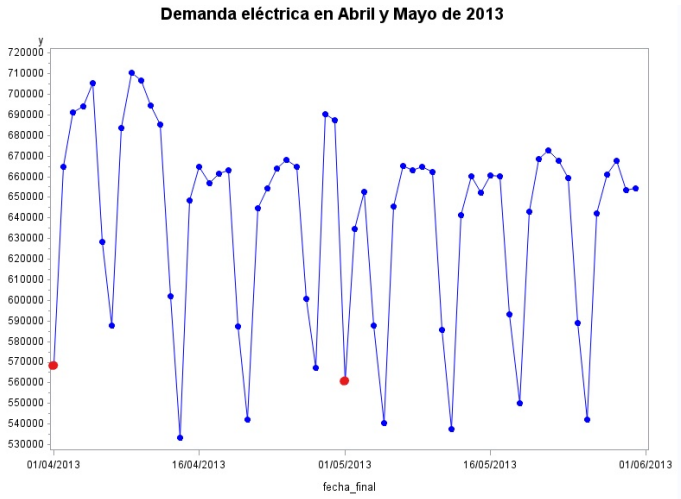


Figura 4.3: Demanda eléctrica nacional en abril y mayo de 2013

Podemos apreciar la estacionalidad semanal de la serie, es decir, un lunes la demanda es parecida al lunes anterior, un martes al martes anterior y así sucesivamente. Si nos fijamos la demanda baja considerablemente en los sábados y aún más los domingos. Esto es debido a la actividad industrial ya que en fines de semana se descansa, especialmente el domingo. Sin embargo, si nos fijamos bien hay dos días (pintados en rojo) especialmente bajos que no son ni sabados ni domingos, de hecho son el lunes 1 de abril de 2013 y el miércoles 1 de mayo de 2013. La explicación a estos datos 'atípicos' es que son festivos, el lunes 1 abril fue un Lunes de Pascua después de Semana Santa (lo que conlleva un periodo vacacional y su consecuente reducción de demanda), mientras que el 1 de mayo es festivo nacional por lo que la industria se para y el día se comporta como el de un domingo. Esto lo vamos a ver dibujando los días 1 de abril y 1 de mayo de 2013 frente a la media de todos los domingos de 2013. También vamos a representar el comportamiento horario según el tipo de día, donde el día puede ser laboral, festivo, postfestivo, prefestivo, sabado o domingo no festivo.

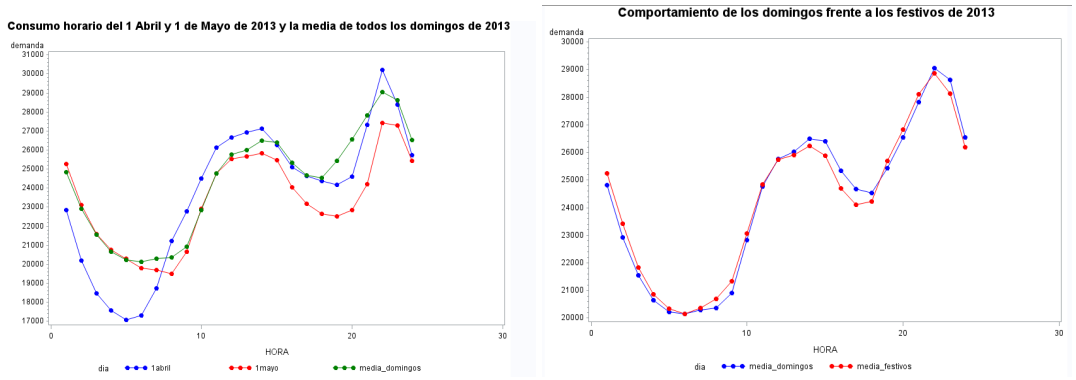


Figura 4.4: Comportamiento horario del 1 abril, 1 mayo, los domingos y los festivos

Como vemos en la imagen de la izquierda el 1 de abril tiene un consumo un poco por encima debido a que no se comporta igual que un festivo nacional ya que en algunas comunidades el Lunes de Pascua se trabaja. Para una mejor comparación, nos fijamos en la imagen de la derecha y vemos el comportamiento de los 9 festivos nacionales de 2013 en su conjunto junto con Viernes Santo y Jueves Santo (color rojo) frente al de los domingos de 2013 (color azul). En esta imagen podemos hacernos una mejor idea de que en media un festivo nacional se comporta como un domingo.
Después de este análisis más intuitivo dibujando la serie, procedemos a un procedimiento más riguroso. Lo primero es evaluar si nuestra serie necesita una transformación, como puede ser la logarítmica o la de la raíz cuadrada. Para ello realizamos el test de Box-Cox para evaluar las distintas transformaciones,

con él obtengo un p-valor de 0,3239179 luego no es necesaria ninguna transformación, (tendría que haber salido un valor muy cercano a 0 para hacer la logarítmica o muy cercano a 0,5 para hacer la transformación con la raíz). Sin embargo, para la red neuronal sí que será necesario meter los datos estandarizados, en nuestro caso se estandarizarán entre -1 y 1.
Por otro lado también es interesante ver la autocorrelación de la demanda, para intuir qué datos de días pasados tienen más correlación con el dato de hoy, para ello dibujamos la gráfica ACF:

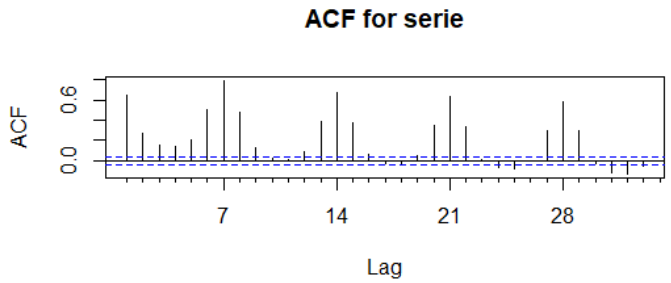


Figura 4.5: Autocorrelación de la serie de demanda

Vemos que el día de hoy en nuestra serie depende mucho de lo que hizo hace una semana, lo cual es lo que habíamos intuído antes, es decir, que para explicar el dato de hoy tenemos que fijarnos en el que hizo el mismo día la semana pasada. Esto lo haremos creando nuevas variables (lags) que sean el valor de hace x días si la serie está a nivel diario. También es recalcable que la demanda de hoy también depende bastante de la demanda que hubo ayer. De hecho la correlación entre la demanda de hoy y la de hace 7 días es de 0,7919809 y entre la de hoy y la de ayer de 0,641912.
Una vez visto la relación entre la demanda y los festivos, la otra variable más importante a la hora del consumo es la temperatura como se ha ido comentando hasta ahora. En las siguientes imágenes podemos ver la relación entre el consumo y la temperatura durante los años 2015 (cuyo verano fue especialmente caluroso) y el pasado, 2017.

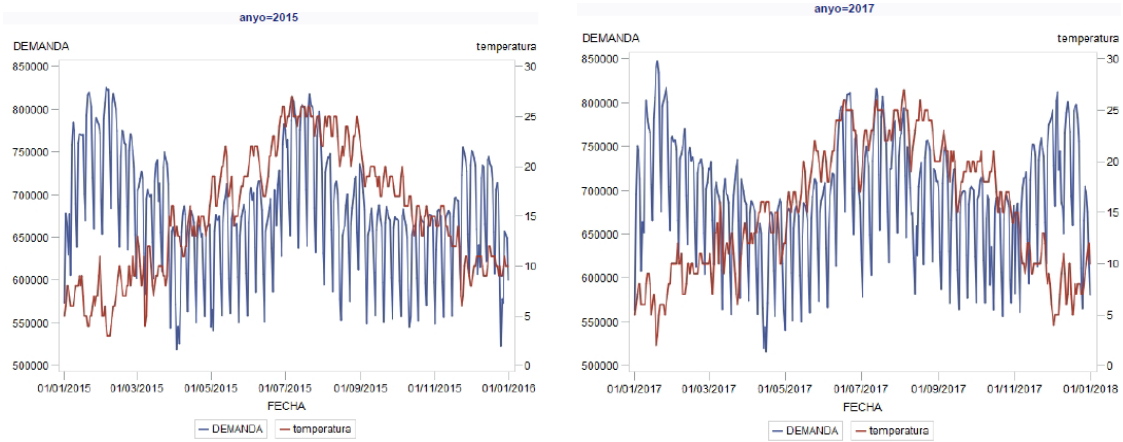


Figura 4.6: Influencia de temperatura en consumo.

Vemos que temperaturas altas en verano hacen que la demanda se incremente mientras que temperaturas bajas en los primeros y ultimos meses del año hacen que vuelvan a crecer los consumos eléctricos. Este patrón se repite todos los años por lo que la variable temperatura debe ser importante a la hora de predecir.
Si, por ejemplo, calculamos una variable que sea el valor absoluto de la temperatura menos su media, es decir:

$$\hat{t} = |t_i - \bar{t}| .$$

$\hat{t}$ representa la distancia de la temperatura a la media, que es la que hace que suba la demanda. Calculamos su correlación con la demanda y obtenemos un valor de 0,50100185 lo cual es muy alto y corrobora esta relación entre temperatura y demanda.

4.1.2. Variables influyentes para la predicción

Una vez visto qué factores pueden influir en la demanda eléctrica, en este apartado explicamos todas las variables que pueden ser útiles para nuestra red (a nivel diario):

- **Fecha**: lo más importante en una serie temporal es la fecha, por lo que a nuestra red tendremos que proporcionar esta variable pero de un modo que la pueda entender bien. Hay que crear variables dummies o binarias que valgan 1 o 0. Por ejemplo, hay que crear 7 variables correspondientes a los 7 días de la semana y 12 meses correspondientes a los 12 meses del año, ya que como vimos el día de la semana y el mes en el que estamos es muy importante para predecir la demanda.
- **Variables de calendario**:
  - **Festivos nacionales**: Otro punto muy importante es el hecho de identificar qué días van a tener un consumo atípico debido a los festivos. Por ello declaramos 9 variables dummies correspondientes a los festivos nacionales para que la red pueda detectarlos. Son variables de tipo pulso.
  - **Festivos CCAA:** Lógicamente no todas las CCAA son igual de importantes en el estudio de la demanda nacional, como es lógico las comunidades con mayor población aportarán un mayor consumo por lo que en un estudio posterior se valorará la posibilidad de incluir algunos festivos de comunidades como variables explicativas, según el impacto que tengan en el consumo. Algunos posibles candidatos serían 2 de mayo (día de Madrid), 28 de febrero (Andalucía), 23 de abril, 25 julio y 19 de marzo que lo tienen varias comunidades, etc.
  - **Periodos Vacacionales:** Hay que tener en cuenta periodos de vacaciones como Navidad, Semana Santa y el mes de agosto. En nuestro caso hemos especificado un poco más creando las variables binarias para la primera y última semana del año, y para las semanas y/o quincenas de agosto. En cuanto a la Semana Santa, se ha probado creando una variable dummy incluyendo desde Jueves Santo hasta el Lunes de Pascua pero también creando 5 variables dummies correspondientes a estos 5 días. Son variables de tipo escalón.
  - **Prefestivo y postfestivo**: Se han introducido las variables binarias de prefestivo, postfestivo y puente ya que la demanda de estos días también se ve afectada aunque en menor medida.
- **Datos meteorológicos**: Se han introducido variables de temperatura máxima y mínima junto con las máximas y mínimas de sensación térmica ponderadas por los pesos de cada Comunidad Autónoma, ya que en general la demanda por comunidades se puede suponer proporcional a la población. Para el estudio horario se introducen las variables por hora de temperatura, luminosidad, viento, humedad y sensación térmica. Estas variables se construyen dando más peso a los datos meteorológicos de las provincias con mayor población, ponderando por el número de habitantes.
- **Demanda**: Es la variable target a predecir. Está a nivel diario (u horario).
- **Variables de retardo**: Para poder hacer ver a la red que la demanda del día de hoy depende de lo que hizo hace una semana, se introducen 7 variables con retardos desde 1 día hasta 7 días, así la red puede utilizar la demanda que hubo la semana pasada, ya que al momento de la predicción estos valores son conocidos.

Hemos creado un archivo con todas estas variables, lo primero que miro es la correlación que tiene cada una con la demanda y:

y	y_1	y_2	y_3	y_4	y_5	y_6	y_7	lunes
1.000000000	0.641912048	0.266866520	0.154072184	0.132310317	0.200945066	0.508584867	0.791980917	0.080481699
martes	miercoles	jueves	viernes	sabado	domingo	enero	febrero	marzo
0.201330534	0.211735062	0.190148016	0.147519067	-0.268472243	-0.562742134	0.205321639	0.259645406	0.019070278
abril	mayo	junio	julio	agosto	septiembre	octubre	noviembre	diciembre
-0.181294179	-0.204681806	-0.029832555	0.147604055	-0.003639671	-0.056703906	-0.177423132	-0.023398449	0.039740561
SS	ind_fiesta	postfestivo	prefestivo	puente	_25dic	_1ene	_1may	_6ene
-0.217872253	-0.240320216	-0.085246386	-0.076281229	-0.040596121	-0.107561195	-0.100012799	-0.110322883	-0.064744255
_6dic	_1nov	_12oct	_8dic	_15ago	agosto__1	domingo_santo	lunes_santo	jueves_santo
-0.016218850	-0.086011007	-0.088284422	-0.031161893	-0.064098286	-0.003639671	-0.122401693	-0.076162984	-0.075111406
viernes_santo	sabado_santo	Tmax	Tmin	SSmax	SSmin	semana1_agosto	semana2_agosto	semana3_agosto
-0.114803317	-0.095840399	-0.118625113	-0.153064236	-0.111696161	-0.154852398	0.044617618	-0.030429478	-0.048223534
semana4_agosto	navidad							
0.020580293	0.012382559							

Figura 4.7: Correlación de las variables explicativas con la demanda

Como vemos los días entre semana tienen una correlación positiva con la demanda lo que indica que en esos días la demanda aumenta, por contra, los sábados y domingos tienen una correlación negativa, luego la demanda disminuye, lo cual tiene sentido. Si nos fijamos en los meses vemos que las correlaciones con enero, febrero, julio y diciembre son positivas luego el consumo crece (meses fríos y verano con la industria en funcionamiento). Ya que cabe destacar que agosto se comporta como un mes normal, ya que aunque sube el consumo por las temperaturas la industria se reduce considerablemente. Si nos fijamos en los que más correlación negativa tienen son abril, mayo y octubre, que son meses en estaciones intermedias, donde no hace ni mucho frío ni mucho calor. Otro punto que da sentido a lo que vimos en las gráficas es que todas las variables binarias construidas para los festivos tienen correlación negativa, esto es que la demanda baja, tomando especial importancia en los días 1 de enero, 25 de diciembre y 1 de mayo. Junto a estos también en los días de Semana Santa vemos que disminuye el consumo.

4.2. Implementación en Python

Una vez visto y analizado cómo es nuestra serie temporal procedemos a crear algunas redes que nos ayuden a predecir la demanda eléctrica durante un año a horizonte 1. Es decir, el objetivo es simular un año entero prediciendo para el día siguiente y así poder obtener un error comparable ya que si lo hiciéramos a nivel mensual el error no es comparable (importa el mes de predicción y si tiene festivos o no y sus temperaturas).
Se presentan los pasos a seguir para su implementación mediante la librería Keras de Python.

4.2.1. Keras

Keras es un interfaz de programación de aplicaciones de redes neuronales de alto nivel escrita en Python. Proporciona una forma rápida y sencilla de definir y entrenar casi todos los tipos de modelos de aprendizaje profundo. Fue desarrollado con un enfoque en permitir la experimentación rápida. Con Keras nos podemos centrar en el diseño de las redes y su entrenamiento dejando un poco de lado los tecnicismos que hay por detrás.
Las principales ventajas que proporciona esta librería son su facilidad de uso, su modularidad, su capacidad de ampliación y que trabaja con Python.
La estructura de Keras es un modelo, una forma de organizar capas. En este trabajo se utilizará el modelo *Sequential* que es una pila lineal de capas, en el que de forma sencilla se van añadiendo capas una detrás de otra.

4.2.2. Pre procesado de los datos

La estandarización de conjuntos de datos es un requisito común para muchos estimadores de aprendizaje automático. Estos métodos pueden comportarse mal si los datos de entrada no están distribuidos normalmente, esto es con media cero y varianza uno.
Las redes neuronales profundas son sensibles a la escala en la que estén los datos de las variables de

entrada especialmente cuando se usan la sigmoidal o la tangente hiperbólica como funciones de activación.
Una estandarización alternativa es escalar las características para que se encuentren entre un valor mínimo y un máximo dado (en la practica se usan los intervalos $[-1,1]$ y $[0,1]$) para que el valor absoluto máximo de cada variable se escale al tamaño de la unidad. Esto se consigue mediante el módulo **sklearn** y su método **MinMaxScaler**. La motivación para usar este escalado incluye la solidez a desviaciones estándar muy pequeñas de las variables. La fórmula explícita que utiliza *MinMaxScaler* para escalar un vector X es, siendo los valores *min* y *max* los valores de entrada entre los que queremos escalar X:

$$X_{std} = \frac{X - min(X)}{max(X) - min(X)}$$
$$X_{escalado} = X_{std} \cdot (max - min) + min.$$

Donde X_{std} está escalado entre 0 y 1 luego $X_{escalado}$ se pone para que esté escalado en el intervalo $[min, max]$ deseado.
La centralización de los datos es muy importante también a la hora de asignar pesos iguales a todas las variables, esto es, que todas las variables sean igual de importantes al inicio de la red. Además hay que escalar todas las variables, tanto las de entrada, explicativas, como las variables a predecir, target. El código es el siguiente:

```
scaler_x=preprocessing.MinMaxScaler(feature_range=(-1,1))
x=np.array(x).reshape((len(x),x.shape[1]))
x=scaler_x.fit_transform(x)
scaler_y=preprocessing.MinMaxScaler(feature_range=(-1,1))
y=np.array(y).reshape((len(y),1))
y=scaler_y.fit_transform(y)
```

4.2.3. Desarrollo de las redes

Una vez preparados los datos para poder ser utilizados procedemos a construir nuestras primeras redes neuronales recurrentes. En nuestro caso vamos a implementar redes usando las células de LSTM (Long Short Term Memory) y GRU (Gated Recurrent Unit). Dentro de cada red hay infinitud de parámetros con los que experimentar hasta obtener mejores resultados ya que se puede jugar con las funciones de activación de cada neurona, con las funciones de pérdida a minimizar, con el número de variables explicativas, con el número de capas y el número de neuronas en cada capa (hay algunas recomendaciones según la topología de la red pero ninguna es 100 % efectiva), etc. En este informe detallaremos las redes que mejores resultados nos han proporcionado.
Antes de centrarnos en cómo construir redes neuronales recurrentes en Python vamos a explicar la técnica del **Dropout**.
El término de *Dropout* se refiere a 'eliminar' o 'perder' neuronas en una red neuronal. Es decir, ignorar un conjunto de neuronas (elegidas aleatoriamente) durante la fase de entrenamiento. De una manera más técnica, el dropout hace que en cada iteración del entrenamiento algunas neuronas no se tengan en cuenta con probabilidad $1-p$ y así se obtiene una red más reducida.
El dropout es necesario para prevenir el sobreentrenamiento (*over-fitting*) ya que una capa totalmente conectada ocupa la mayor parte de los parámetros y por tanto las neuronas desarrollan codependencia entre ellas, lo que restringe la potencia individual de cada una y produce este sobreentrenamiento. En la imagen siguiente se ve cómo queda una neurona tras aplicar este método:

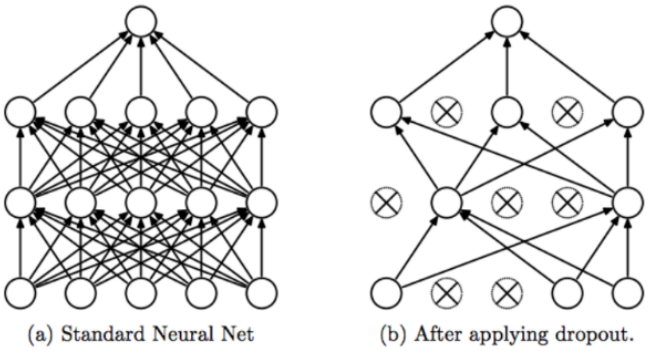


Figura 4.8: Consecuencias del *Dropout*

No hay un valor óptimo de probabilidad p para el dropout pero después de pruebas preliminares se puede concluir que el valor 0,2 es el que mejor funciona.
Una vez explicada esta técnica que usaremos (se probó sin utilizarla y se obtuvieron peores resultados por lo que se decidió utilizar esta técnica), los modelos de redes que implementaremos serán sencillos y con pocas capas. Como ya hemos dicho, se utiliza el modelo *Sequential* para crear las redes. Además todas nuestras redes tendrán la capa de entrada en la que se indica el número de variables explicativas x que entran al modelo y la capa de salida con una única neurona que hace referencia a nuestra predicción de demanda y.

- **Red Neuronal Artificial**:
  Esta primera red neuronal artificial (no recurrente) consta de una capa de entrada con 100 neuronas a la que se le aplica un dropout, luego se le añaden dos capas intermedias más de 1 y 64 neuronas respectivamente para acabar con una capa de salida formada por una neurona. A este modelo se le aplica el método de *Stochastic Gradient Descent* donde se usan los parámetros explicados anteriormente como un learning rate de 0.01 y un momentum de 0.9.

Listing 4.1: Red Neuronal Artificial Simple

```
model = Sequential()
model.add(Dense(100,
                input_shape=(x_train.shape[1],),
                use_bias=True))
model.add(Dropout(0.2))
model.add(Dense(1))
model.add(Dense(64, activation='tanh'))
model.add(Dense(1, activation='tanh'))
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mae', optimizer='adam')
model.summary()
history = model.fit(x_train, y_train,
                    epochs=300, batch_size=100,
                    validation_data=(x_val, y_val),
                    shuffle=True)
```

Con el comando *summary* podemos ver la estructura de la red de forma resumida y el número de parámetros:

Tipo de capa	Forma de la salida	Número parámetros
Densa	(None, 100)	4200
Dropout	(None, 100)	0
Densa	(None, 1)	101
Densa	(None, 64)	128
Densa	(None, 1)	65

Cuadro 4.1: Estructura red multicapa simple sin bias. Total de parámetros: 4494.

- **Célula LSTM**:
  El primer modelo recurrente que realizamos es uno muy sencillo, con dos capas intermedias, una célula LSTM y otra densa. En este primer modelo el término independiente sesgado sí lo ponemos (use_bias=True).

Listing 4.2: LSTM 1 con bias

```
model = Sequential()
model.add(LSTM(100,
               input_shape=(x_train.shape[1], 1),
               use_bias=True))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss='mae', optimizer='adam')
model.summary()
```

La estructura de esta red es:

Tipo de capa	Forma de la salida	Número parámetros
LSTM	(None, 100)	40800
Dropout	(None, 100)	0
Densa	(None, 1)	101

Cuadro 4.2: Estructura LSTM con bias. Total de parámetros: 40901.

- **Célula GRU**:

Listing 4.3: GRU 1 con bias

```
model = Sequential()
model.add(GRU(output_dim=100,
              return_sequences=False,
              activation='tanh',
              inner_activation='hard_sigmoid',
              input_shape=(x_train.shape[1], 1),
              use_bias=True))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss='mae', optimizer='adam')
model.summary()
```

El número de parámetros con una célula GRU es menor que los de la LSTM debido a que esta célula tenía menos puertas.

Tipo de capa	Forma de la salida	Número parámetros
GRU	(None, 100)	30300
Dropout	(None, 100)	0
Densa	(None, 1)	101

Cuadro 4.3: Estructura GRU con bias. Total de parámetros: 30401.

Como hemos visto en los códigos, en mi caso opto por usar como función de pérdida para la red el error absoluto medio (MAE) aunque a la hora de comparar modelos se utilizará el error absoluto porcentual medio (MAPE). Otro parámetro con el que se puede jugar es el optimizador, en nuestro caso usamos el optimizador *adam*. El error absoluto medio se construye a partir de la siguiente expresión:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}$$

donde e_i es el error cometido para la observación i, es decir, el valor predicho y_i menos el valor real x_i en valor absoluto y n es el número de observaciones predichas.
Una vez creada nuestra red lo que tenemos que hacer es entrenarla. Para ello todo el histórico de datos del que disponemos (del 1 de enero de 2012 al 31 Mayo de 2018) se reparte en los conjuntos de entrenamiento, validación y test. Como queremos predecir un año (del 1 junio de 2017 al 31 de mayo de 2018) estos datos serán los de test. Por otro lado usamos como datos de entrenamiento del 1 enero de 2012 al 31 de mayo de 2016 y dejamos desde el 1 de junio de 2016 al 31 de mayo de 2017 para validación. Por tanto para entrenar el modelo se introduce la siguiente sentencia:

Listing 4.4: Entrenamiento del modelo

```
history = model.fit(x_train, y_train,
                    epochs=300, batch_size=100,
                    validation_data=(x_val, y_val),
                    shuffle=True)
```

Batch_size es el número de muestras con las que se va actualizando el gradiente. *Epochs* es el número de épocas para entrenar el modelo, siendo una época una iteración sobre el total de datos y *shuffle=True* indica que los ejemplos de entrenamiento se mezclan aleatoriamente en cada época, esta técnica ayuda a la red a aprender.
A la hora de comparar modelos se ha puesto que el tamaño del batch sea de 100 para una mayor rapidez puesto que 300 épocas con batch 100 tarda entrenar la red unos 10 minutos y se han hecho muchas comparaciones según los parámetros y según qué variables explicativas deben entrar. Sin embargo, una vez elegida la red, se realiza el entrenamiento con batch 10 para una mayor precisión en el entrenamiento.

4.2.4. Backtesting

El *Backtesting* es el proceso de testear una estrategia antes de emplearla. Esto nos permite conocer cómo de bueno es el modelo que estamos usando para predecir. El objetivo del backtesting es evaluar la eficacia de nuestra táctica y sistema frente a otros.
La forma rigurosa de su realización sería simular la predicción de 1 año a horizonte 1 y luego comprobar con los datos de test, esto es, empezamos prediciendo para el 1 de junio de 2017 con todos los datos hasta el 31 de mayo de 2017, y así para todos los días, por ejemplo,para predecir el día 20 septiembre de 2017 se podrán utilizar datos de entrenamiento hasta el valor real del día 19 de septiembre (como

si la predicción la hiciéramos el 20 de septiembre con todos los datos disponibles). Esto se debe hacer durante un año ya que por las características de nuestra serie no es comparable el error que se puede cometer al predecir junio que al predecir diciembre del mismo año, puesto que junio no tiene ningún festivo y en diciembre se juntan muchos, por lo que es más probable un MAPE más alto en diciembre.
Sin embargo para su realización son necesarias entrenar 365 redes neuronales (una por cada día, aumentando cada vez el conjunto de entrenamiento) pero por falta de recursos se realizará un test en el que se predecirá un año con el mismo conjunto de entrenamiento pero con las variables de retardo se tendrán los datos reales de los últimos 7 días.
Para puntuar un modelo usaremos el MAPE que es:

$$MAPE = \frac{100\,\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - x_i}{x_i} \right|$$

donde y_i vuelve a ser el valor estimado y x_i el valor real de la observación i.

Capítulo 5

Comparación de modelos

A la hora de obtener resultados se han desarrollado otros modelos aparte de las redes neuronales recurrentes con el objetivo de poder comparar y concluir si las redes neuronales recurrentes mejoran las predicciones de otros tipos de modelos.
Las opciones para predecir que se han tenido en cuenta para la comparación son: no tener modelo, modelo ARIMA/SARIMA y redes neuronales. Esta comparación es muy útil ya que obtener por ejemplo un error MAPE del 3 % con redes no significa nada sin saber qué error obtienen otros modelos. Imaginemos que sin un modelo se obtiene un error del 3 % también, esto nos indicaría que el modelo de redes que hemos desarrollado anteriormente no mejora nada luego sería un modelo malo. A continuación presentamos los resultados obtenidos con distintos métodos de predicción a horizonte 1. Los datos de test son un año que va desde el 1 de junio de 2017 hasta el 31 de mayo de 2018 y las variables explicativas que entran en esta comparación (las mismas para todos los modelos) son la demanda real de días anteriores, los 9 festivos nacionales junto con la variable binaria de Semana Santa y las temperaturas máximas y mínimas diarias.

- **Observación:** Hay que tener en cuenta que para poder comparar modelos es necesaria la reproducibilidad de los mismos, ya que muchos modelos, como por ejemplo las redes, necesitan de variables aleatorias que hacen inicializar pesos u otras funciones. Para que los resultados sean reproducibles se han fijado las semillas aleatorias con las que se iniciaban los modelos.

5.1. Predicción sin modelo

Sin saber nada sobre ningún modelo matemático de predicción y habiendo estudiado cómo se comporta la serie se nos pueden ocurrir tres formas de predecir la demanda sin usar nada de índole matemática. La primera y más sencilla es dar como predicción el valor del día anterior. Si pensamos un poco en el comportamiento estacional semanal de la serie se nos podría ocurrir como predicción el valor del mismo día de la semana anterior (el valor de hace 7 días) ya que como vimos se podría considerar un ciclo semanal en la serie. Por último también podríamos pensar en dar el valor de ese mismo día del año pasado, es decir, el valor de hace 365 días (sin año bisiesto entre medio). Los resultados que se obtienen con estas opciones son:

- Usando el dato de ayer como predicción para hoy. Esto se ha implementado con la función *shift(1)* en Python y el error MAPE obtenido es de un 6,319229 %.

- Usando el dato de hace una semana. Para calcularlo se ha vuelto a usar la función *shift(7)* y el error obtenido ha sido un MAPE de 4,9523726 %.

- Usando el dato de hace un año. Se construye de la misma manera que las otras y se obtiene un error absoluto medio porcentual de 7,3797515 %.

Vemos que la mejor predicción sin un modelo matemático consistiría en coger el valor del mismo a predecir de la semana anterior. Este modelo mejora al de coger el día anterior como es lógico ya que en este último modelo los lunes se cogería el valor del domingo anterior, por lo que en la mayoría de los lunes el error sería muy grande. Sin embargo, coger el valor respecto hace un año puede ser peligroso ya que es un periodo de tiempo muy grande y ha podido haber cambios importantes en el consumo (además de que los días de la semana no coinciden). Por lo que el mejor error obtenido sin tener modelo es un MAPE del 4,9523726 %.
Si hacemos un estudio sobre los días que más error cometemos con estos 'modelos' vemos que en media los días que más error se comete son los lunes, sábados y domingos, debido a los saltos comentados anteriormente. En las siguiente tabla y gráficas se puede observar mejor:

Día semana	MediaMAPE1	MediaMAPE7	MediaMAPE365
lunes	15,097	5,615	16,743
martes	3,667	5,423	5,969
miércoles	2,453	4,734	3,986
jueves	2,127	5,196	4,211
viernes	1,695	5,509	4,566
sábado	11,151	4,337	9,443
domingo	8,125	3,847	6,802
global	6,319	4,952	7,380

Cuadro 5.1: Media de errores en cada día según el modelo.

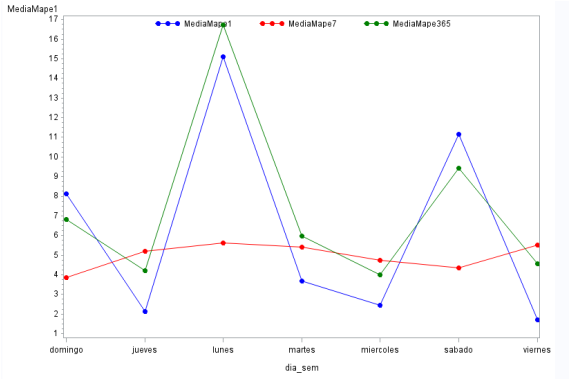


Figura 5.1: Días con mayor error sin un modelo

5.2. Predicción con ARIMA/SARIMA

Otro método muy conocido a la hora de predecir son los modelos autorregresivos y de medias móviles ARIMA/SARIMA. Casi parece obligatorio presentar alguna predicción con estos modelos para poder evaluar la mejora que introducen las redes frente a estos modelos. En este trabajo no se explicará la teoría detrás de estos modelos por lo que se explicará por encima su implementación y se presentarán los resultados obtenidos. Dentro de este apartado vamos a evaluar tres modelos. El modelo ARIMA utilizado será el que nos diga la función *auto.arima* de *Rstudio*. Los otros dos serán modelos SARIMA,

uno el que nos diga la función *auto.arima* de R y el otro se obtiene teniendo en cuenta las gráficas de autocorrelación parcial (PACF) y total (ACF).

- ARIMA correspondiente a la función auto.arima: Se ha introducido nuestra serie de demanda eléctrica en el software de *Rstudio* y se ha aplicado la función de auto.arima después de haber preparado los datos. El ARIMA que nos ha seleccionado ha sido el ARIMA(2,1,2) es decir el correspondiente a la siguiente expresión:

$$(1-\phi_1 B-\phi_2 B^2)(1-B)X_t = \mu + (1-\theta_1 B-\theta_2 B^2)a_t$$

  donde B es el operador matemático diferencia, es decir $BX_t = X_{t-1}$ y $B^2X_t = X_{t-2}$. Con este modelo se obtiene un error MAPE de 5,510087 %.

- SARIMA correspondiente a la función auto.arima: Dentro de la función auto.arima también es posible pedir que nos de un ARIMA con componente estacional, es decir, un SARIMA. El SARIMA resultante ha sido el SARIMA$(4,1,0)x(2,0,0)_7$. En formulación matemática sería:

$$(1-\phi_1 B-\phi_2 B^2-\phi_3 B^3-\phi_4 B^4)(1-B)(1-\Phi_7 B^7-\Phi_{14} B^{14})X_t = \mu + a_t.$$

  Con este modelo el error MAPE obtenido es 2,7011937 %. Vemos que la componente estacional en esta serie es muy importante y su inclusión hace que baje bastante el error.

- SARIMA obtenido tras el análisis de las gráficas ACF y PACF: Tras un análisis mirando las autocorrelaciones totales y parciales de las variables, se ha conseguido que los residuos sean ruido blanco con el SARIMA$(1,0,2)x(1,1,1)_7$ por lo que el mejor modelo es el siguiente:

$$(1-\phi_1 B)(1-\Phi_7 B^7)(1-B^7)X_t = \mu + (1-\theta_1 B-\theta_2 B^2)(1-\Theta_7 B^7)a_t.$$

  Con este modelo un poco más sofisticado que los anteriores se ha obtenido un MAPE de 1,6306 %. Dentro de los modelos ARIMA/SARIMA destacamos que el modelo obtenido tras el análisis de las autocorrelaciones es el que mejor predice a horizonte 1 durante un año entero, obteniendo un MAPE de sólo 1,6306 %. Vemos que se ha mejorado la predicción que obtuvimos sin modelo.

5.3. Predicción con redes neuronales

En este apartado se han implementado las redes neuronales desarrolladas en el apartado anterior; una red neuronal artificial (RNA) y dos redes neuronales recurrentes (RNR) mediante las células LSTM y GRU. Presentamos los resultados obtenidos:

- RNA: Mediante la implementación de la red neuronal artificial como la explicada antes se obtiene un MAPE de 1,76443471 %, como vemos no llega a mejorar al error obtenido mediante el segundo SARIMA. Si nos fijamos en los días que obtenemos un mayor error nos damos cuenta que los días que predecimos bastante menos de lo que se consume son el 2 de enero, 2 de noviembre, 26 de diciembre, 2 de mayo (todos postfestivos). Sin embargo los días que se predice más de lo que se consume son el 1 de noviembre, el 6 de diciembre, 30 de abril, etc. Esto nos indica que en los festivos se sigue quedando por encima del consumo y en los postfestivos se queda por debajo, por lo que deducimos que le cuesta aprender este comportamiento, aunque hay que tener en cuenta que las variables de postfestivo y prefestivo no se han introducido para esta comparación. Además es importante mencionar que el entrenar esta red nos ha llevado menos de 5 minutos.

- RNR LSTM: Si en vez de una red neuronal artificial predecimos con la red recurrente explicada antes usando una célula LSTM obtenemos un error MAPE del 1,60677611 %. Por poco pero con esta red ya hemos conseguido bajar el error de nuestro mejor modelo hasta ahora. En este caso los días con un error mayor son: el 29 de marzo, el 1 de enero, el 26 de diciembre, el 2, 3 y 8 de

enero, pero con un error un poco menor que el anterior.
En la siguiente tabla podemos ver cómo ha ido mejorando la predicción según se iban incluyendo variables explicativas a la red LSTM. Los parámetros fijos son *shuffle=true*, *loss function=mae*, *epochs=300* y *optimizer=adam*:

Variables	Batch_size	Error MAPE
dia+mes	100	4,7599
dia+mes+7lags	100	2,3087
dia+mes+7lags+fiest	100	1,9823
dia+mes+7lags+fest+prepostpuente+temp	100	1,6948
dia+mes+7lags+festivos+temp	10	1,6068

Cuadro 5.2: Errores con distintos modelos LSTM

El tiempo de entrenamiento de la red LSTM con mejor resultado y *batch_size*=10 ha sido de unos 30 minutos.

- RNR GRU: Por último usamos nuestra otra red recurrente, desarrollada con una célula GRU. Este modelo es más rápido que el anterior debido a que tiene menos parámetros. El error obtenido es un MAPE de 1,4101461 %. La predicción durante un año se ha conseguido rebajar casi un 15 % frente a los modelos SARIMA lo cual es un paso importante, además teniendo en cuenta que con errores tan bajos cada vez es más difícil mejorarlos. En este mejor modelo los días que más error hemos cometido han sido: el 2 de enero, el 13 de octubre, el 1 de enero y el 30 de marzo pero ya con una MAPE mucho menor al de los errores de la RNA. Además el tiempo de entrenamiento ha sido menor que el de LSTM, siendo este de unos 20 minutos.

Hemos visto que las redes neuronales han mejorado la predicción que se obtiene sin tener un modelo e incluso frente a los modelos ARIMA/SARIMA que son los más utilizados hoy en día. También cabe destacar que en los días donde más se ha equivocado la red son días que se presuponen que hay más cambio respecto al consumo normal. Estos días son pre y post festivos, semana santa y días justo después de periodos vacacionales como navidad, por lo que intuímos que aun es mejorable si incluyemos las variables explicativas idóneas. En la siguiente imagen vemos el año de test de nuestro mejor modelo, siendo la roja la actual y azul la predicción:

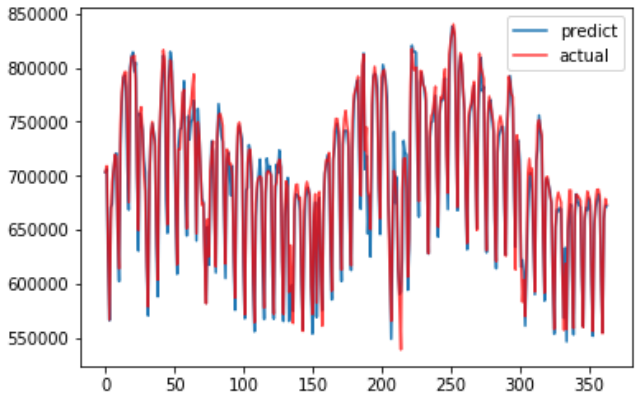


Figura 5.2: Predicción demanda con RNR con célula GRU.

El siguiente apartado tratará de intentar mejorar este modelo obtenido mediante la inclusión de nuevas variables y la predicción a nivel horario con redes.

Capítulo 6

Resultados en redes neuronales recurrentes

En este capítulo vamos a ver y comparar dos formas de predecir a nivel horario. Por otro lado, como en el capítulo anterior hemos visto que las redes neuronales mejoran los otros modelos, nos proponemos mejorar los resultados obtenidos hasta ahora en la predicción a nivel diario.

6.1. Predicción horaria

Una vez estudiado la demanda a nivel diaria, tratamos realizar una predicción a nivel horario, es decir, más fina que la anterior. Para ello compararemos dos formas de predecir la demanda a nivel a horario:

1. Predecir la serie a nivel diario (como en el capítulo anterior) y luego estimar la proporción de consumo horario y desagregar.

2. Predecir directamente la demanda a nivel horario.

La idea es comparar estas dos metodologías de predicción de demanda eléctrica. A priori dan mejores resultados la primera estrategia debido a que muchas variables explicativas como los festivos, vacaciones, etc, son a nivel diario y, además, por otro lado la proporción de consumo horario suele seguir un patrón muy estable y es fácil de predecir.

6.1.1. Predicción de la proporción diaria y desagregación

Para este caso, utilizaremos la predicción obtenida con la red neuronal recurrente GRU ya que es con la que obtenemos un MAPE menor (1,4101461 %). Tras un estudio de la proporción de consumo horario se llega a la conclusión de que la mayoría de días laborables siempre se comportan igual, mientras que los sábados es un poco distinta y los domingos y festivos más diferente (aunque entre ellos parecida como vimos en 4.4). En el siguiente gráfico podemos ver cómo se comporta la proporción horaria en media del último año (si cogiésemos más años es muy parecido) en cada día, yendo los festivos en un grupo distinto.

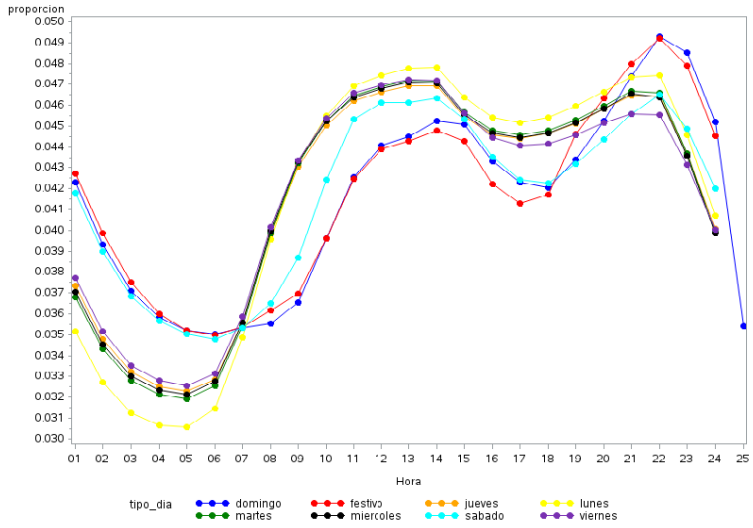


Figura 6.1: Proporción consumo horario según el tipo de día.

Como vemos la proporción es muy parecida si el día es laborable y también entre domingos y festivos, quedándose el sábado con un comportamiento a medias entre laborables y domingos. Tenemos esta proporción en una tabla como la siguiente:

HORA	domingo	festivo	jueves	lunes	martes	miercoles	sabado	viernes
01	0.0423258612	0.042708523	0.0373276313	0.0351440947	0.0367894292	0.0370336041	0.0417837512	0.0377060014
02	0.0393377472	0.0398745679	0.0347825291	0.0327223413	0.0343161131	0.0345365294	0.0389973463	0.035176181
03	0.0370897667	0.0374992546	0.0332068968	0.0312705056	0.0328020056	0.0329969135	0.0368473363	0.0335405969
04	0.0358507106	0.0360002991	0.0325315578	0.0306461923	0.0321356361	0.0323400285	0.0356641377	0.0328140049
05	0.0351873356	0.0351986366	0.0322962092	0.0305617848	0.0319380044	0.032130997	0.0350349937	0.0325707239
06	0.035042618	0.03498221	0.0328595443	0.0314606342	0.0325708257	0.0327629444	0.0347921299	0.0331544922
07	0.0353166108	0.0353792145	0.035571363	0.0348725399	0.0354217464	0.0355755745	0.0353066781	0.0358836634
08	0.0355494195	0.0361511008	0.0398429811	0.0395690227	0.0398821036	0.0400055598	0.0364915387	0.0401604021
09	0.0365594681	0.036963279	0.0430627352	0.0432115018	0.0432479253	0.0432953353	0.0387083333	0.0433407039
10	0.039594367	0.0396218678	0.0450447934	0.0455202122	0.0452689697	0.0452579734	0.0424399566	0.0453706749
11	0.0425445809	0.0424627449	0.0462217949	0.0469336192	0.0464642277	0.0463969069	0.0453207291	0.0465888566
12	0.0440590471	0.043918895	0.0466319381	0.0474267253	0.0468698905	0.0467850291	0.0461138802	0.0469858837
13	0.044511104	0.0442892493	0.0469403585	0.0477519959	0.0471732851	0.0470828819	0.0461225642	0.0472264639
14	0.0452338845	0.0447989131	0.0469381786	0.0478137529	0.0471541885	0.0470571283	0.0463278815	0.0471766571
15	0.0450997452	0.0442728596	0.0454598698	0.0464008699	0.0456598224	0.045558771	0.0453285354	0.0456150611
16	0.0433293021	0.0422361654	0.0445741996	0.0454370138	0.0447673231	0.0446681265	0.0435334286	0.0444342493
17	0.0422864837	0.0413105154	0.0444111094	0.0451798397	0.0445618019	0.0444568692	0.0424244339	0.0440539845
18	0.0420640595	0.0417356864	0.0446600635	0.0454127267	0.0448056069	0.0446767495	0.0422465486	0.0441380168
19	0.0434095783	0.0446275325	0.0451358021	0.0459436563	0.0452892904	0.0451518339	0.0431759825	0.0445602443
20	0.0452383275	0.0463210333	0.0458315387	0.0466540537	0.0459729827	0.0458526229	0.0443643918	0.0451749832
21	0.0473980016	0.0479795685	0.0464720378	0.0473621014	0.0466738683	0.0465318974	0.0456041491	0.0456039302
22	0.0492893085	0.0492092346	0.0464349414	0.0474325942	0.0465989588	0.0463997389	0.0464973062	0.0455261302
23	0.0485157231	0.0479093275	0.0437017118	0.0445675935	0.0437064817	0.0435617054	0.0448722252	0.0431514639
24	0.0451996066	0.044549321	0.0400602147	0.0407046281	0.039929513	0.0398842794	0.0420017418	0.0400466307

Figura 6.2: Tabla de proporciones horarias.

Ya sólo queda multiplicar la demanda diaria que es nuestro vector salida de la red por la proporción de consumo en cada hora para así obtener una predicción a nivel horario. Una vez obtenida esta predicción se compara con la demanda horaria real que hubo y obtenemos un error MAPE del 2,7950858 %. Como vemos ha subido bastante en comparación con el diario.

6.1.2. Predicción directa a nivel horario

La otra opción que se ha comentado es la predicción de la demanda eléctrica directamente a nivel horario. Para su realización se necesitan datos de consumo al mismo nivel durante un año, por lo tanto,

la longitud de los datos ahora se multiplica por 24. Esto hace que los recursos necesarios para su entrenamiento sean mucho mayores, principalmente los tiempos de ejecución se han multiplicado.
Esta predicción a nivel horario se ha hecho tanto con el modelo con la célula GRU como con la célula LSTM. Además se han usado las mismas variables explicativas, es decir, los festivos nacionales y la temperatura, obteniendo los siguientes resultados:

- GRU: MAPE de 3,87672403 %.
- LSTM: MAPE de 4,22370379 %.

Como vemos obtenemos un peor resultado al predecir directamente a nivel horario, tanto con la célula LSTM como con la GRU, que predecir a nivel horario y luego desagregar. De hecho, este error ha aumentado en más de un 1 %

6.2. Mejoras en predicción diaria

Como acabamos de ver, a la hora de predecir a nivel horario es mejor predecir diariamente y luego desagregar. Por tanto vamos a partir de nuestra mejor predicción diaria obtenida en el apartado 5.3, donde, usando célula GRU, obtuvimos un error MAPE de 1,4101461 %.
Estudiando los errores que cometemos cada día vemos que la mayoría de los días con un error mayor son los correspondientes a algunos postfestivos. Por tanto planteamos la introducción de una variable que indique cuándo un día es un postfestivo o no.
Con la introducción de la nueva variable obtenemos un error MAPE de 1,36308456 %. Si nos volvemos a fijar en los días en los que más error cometemos vemos que muchos se concentran en la última semana del año, por ello planteamos introducir una variable que valga 1 cuando el día está comprendido entre el 24 de diciembre y el 31 de diciembre (ambos incluidos). Con estas nuevas variables explicativas conseguimos volver a reducir el error, obteniendo un MAPE de 1,298594 %.
Nos seguimos fijando en los días con un mayor error. En este caso nos encontramos con días como el 11 de septiembre, el 2 de mayo y el 25 julio entre otras. Con un estudio de estas fechas nos damos cuenta que se corresponden con el día de Cataluña, el día de Madrid y el día de Galicia luego introducimos estas variables dummies al modelo, obteniendo un error MAPE de 1,52487281 %. Como vemos la inclusión de variables no siempre implica una mejora en los resultados. Lógicamente los días concretos han mejorado pero el error medio ha subido debido a las distintos pesos de la nueva red.
Un resumen de los resultados obtenidos con la célula GRU y parámetros *shuffle=true*, *loss function=mae*, *batch_size=10*, *epochs=300* y *optimizer=adam*, es decir, sólo cambiando las variables explicativas es el siguiente:

Variables	Error MAPE
dia+mes	5,13343
dia+mes+7lags	1,95047
dia+mes+7lags+festivos	1,49949
dia+mes+7lags+festivos+temp	1,41015
dia+mes+7lags+festivos+temp+postfestivo	1,36308
dia+mes+7lags+festivos+temp+postfestivo+ultima_semana	1,29859
dia+mes+7lags+festivos+temp+postfestivo+ultima_semana+3ccaa	1,52487

Cuadro 6.1: Errores con distintos modelos GRU

En la tabla vemos los resultados obtenidos con distintos modelos, siendo el mejor con las variables explicativas del día de la semana, del mes, los festivos nacionales, los 7 valores de los días anteriores, la

temperatura máxima y mínima diaria, la variable postfestivo, la variable de Semana Santa y la inclusión de la última semana del año, obteniendo un error de 1,298594 %.
Con el estudio de la serie y de los resultados obtenidos por cada modelo podemos estudiar qué variables son importantes a la hora de predecir la demanda y valorar su introducción como variables explicativas para reducir su error, aunque no siempre se mejora. Aún así, con un estudio sencillo de los días que más error se comete hemos conseguido reducir el error un poco más de $0,1\,\%$ lo cual con estos resultados es bastante.

Capítulo 7

Conclusiones

Este trabajo ha servido para ampliar los conocimientos aprendidos en las asignaturas de Técnicas de Predicción y de Estadística Aplicada y Minería de Datos del máster. Gracias a estas asignaturas principalmente se han asentado las bases de esta investigación en las técnicas de las redes neuronales para la predicción.
Tras este proyecto se han estudiado de forma teórica las redes neuronales y cómo funcionan (en especial las recurrentes). Además se ha realizado un análisis de la demanda eléctrica nacional y se han comparado diferentes metodologías de predicción. Después de todo el trabajo llegamos a las siguientes conclusiones:

- Machine Learning y Deep Learning son dos sistemas capaces de aprender solos, a comparación con todas las demás tecnologías anteriores. En particular, el Deep Learning va más allá de lo que llega el Machine Learning siendo capaz de aumentar la precisión de las conclusiones/resultados del problema a tratar.
- Ambos sistemas necesitan de grandes cantidades de datos en los que fijarse para poder aprender.
- Se han visto la mayoría de características de las redes neuronales. Sabiendo clasificar una neurona según su topología o su aprendizaje. Además se han estudiado las etapas de aprendizaje y la forma en la que entrenan las redes.
- Se han introducido las redes neuronales recurrentes, estudiando en qué se diferencian respecto a las artificiales y viendo cómo es su funcionamiento en concreto en su proceso de memoria.
- Se ha estudiado la serie de demanda eléctrica nacional de manera analítica, viendo su estacionalidad semanal y su comportamiento frente a festivos, periodos vacacionales y temperatura.
- Se han implementado en el software de Python modelos de redes neuronales recurrentes sencillos con el objetivo de predecir la serie de la demanda. También se han utilizado modelos ARIMA y SARIMA para esta misma predicción.
- Se han comparado el no tener modelo, los modelos ARIMA/SARIMA y las redes neuronales como modelos de predicción.
- Se ha desarrollado un modelo de backtesting para que la comparación anterior sea efectiva. Se ha utilizado el MAPE.
- Se ha comprobado que con modelos sencillos de redes neuronales recurrentes se mejoran los modelos actuales de los que disponemos. Por lo que sería interesante en un futuro seguir investigando estas redes para su mejora.
- Se ha concluido que es mejor predecir a nivel diario y desagregar que predecir directamente a nivel horario.

Posibles mejoras en un futuro

En cuanto a posibles mejoras en los modelos realizados principalmente se podrían valorar:

- Mejora en el cálculo de la proporción horaria. Por ejemplo mediante la implementación de redes que calculen la proporción de consumo horaria según el tipo de día. Una clasificación sencilla de los tipos de días que se comportan distinto podría ser:
  - Viernes.
  - Sabados.
  - Festivos (domingos incluidos).
  - Laborables post-festivos.
  - Laborables no post-festivos.
- Construir redes más complejas con más capas que puedan entender relaciones no lineales.
- Jugar con los parámetros de las redes construidas. Por ejemplo se pueden modificar:
  - Función de pérdida.
  - Optimizador.
  - Learning rate.
  - Momentum.
  - Dropout.
  - Número de épocas y tamaño del batch.
  - etc.
- Estudiar introducir las variables de temperatura ponderada por población y por consumo horario en lugar de las temperaturas máxima y mínimas diarias.
- Estudio de los días en los que más se equivoca la red y entender por qué. Una vez entendido, valorar la inclusión de nuevas variables para que la red aprenda que esos días son distintos a un día normal.

Bibliografía

[1] ARMANDO FANDANGO, *Python Data Analysis- Second Edition*, Packt Publishing, Birmingham, marzo 2017.

[2] AURELIÉN GÉRON, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, O'Reilly Media, Sebastopol, marzo 2017.

[3] FRANÇOIS CHOLLET, *Deep learning with Python*, `http://www.deeplearningitalia.com/wp-content/uploads/2017/12/Dropbox_Chollet.pdf` Manning Publications, Nueva York.

[4] IAN GOODFELLOW, YOSHUA BENGIO, AARON COURVILLE, *Deep Learning*, `http://www.deeplearningbook.org`, MIT Press, 2016.

[5] N. D LEWIS, *Deep time series forecasting with Python*, diciembre, 2016.

[6] KERAS, *Keras: The Python Deep Learning library*, `https://keras.io/`.

[7] KAGGLE, *Kaggle*, `https://www.kaggle.com/`.

[8] DANIEL VÉLEZ SERRANO, CARLOS RIVERO, MARÍA DEL CARMEN PARDO, *Material de la asignatura Estadística Aplicada y Minería de Datos*, Universidad Complutense de Madrid.

[9] ELENA ROSA PÉREZ, FEDERICO LIBERATORE, *Material de la asignatura Técnicas de Predicción*, Universidad Complutense de Madrid.