ANEXOS

ANEXO 1: TIPOS DE REDES NEURONALES[4].

Las redes neuronales se dividen en diferentes tipos según su estructura y funcionamiento pudiendo diferenciarlas en tres tipos más comunes:

-Red Neuronal Profunda (Deep Neural Net: DNN):

Esta red está formada por distintas capas neuronales, teniendo cada una de ellas un número determinado de neuronas y dividiéndose en tres tipos principales, como indica la *Figura 1.1*.

Una capa de entrada en la cual introducimos los datos que tenemos.
Una o varias capas ocultas, las cuales establecen las distintas relaciones entre las distintas variables de nuestro modelo.

Y por último una capa de salida, la cual realiza la predicción del resultado final según los distintos valores de las variables de la capa de entrada y las correlaciones que se hayan dado en las capas ocultas.

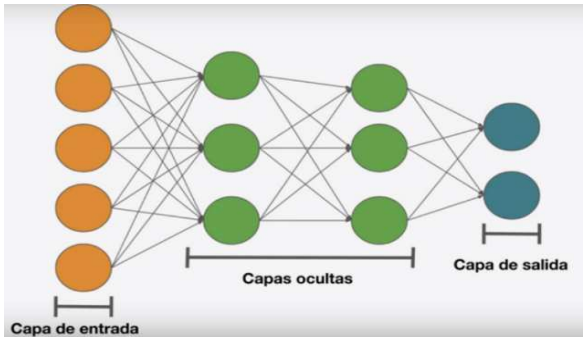

Figura A1.1: Representación de una red neuronal profunda: DNN

-Red Neuronal Convolucional[5] (CNN):
Usualmente utilizada para el procesamiento de texto, pero su uso más común es el procesamiento o reconocimiento de imágenes.
Tiene una estructura similar a la anterior, pero en la capa oculta se realizan operaciones de *convoluciones*, para obtener características importantes de la imagen de entrada como detección de bordes, afilado, reconocimiento de figuras, desenfoque reconocimiento de determinados elementos y *max-poling*[6] *o* conversión de la imagen en otra más simple pero que siga teniendo las características más importantes.

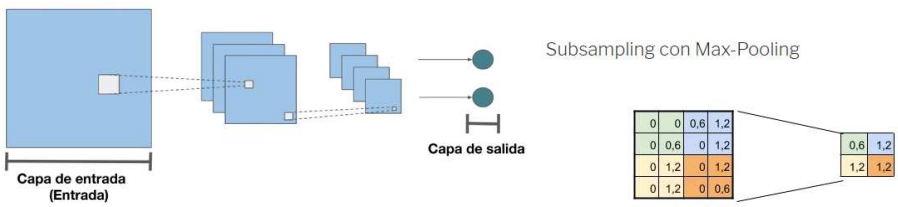

Figura A1.2: Representación esquemática de los dos procesos principales que se dan en las redes convolucionales que son el proceso de *convolución* y *max-pooling.*

-Red Neuronal Recurrente (RNN):
Esta se utiliza cuando los datos con los que operamos son secuenciales, lo que significa que hay una correlación temporal entre ellos, utilizándose mucho para texto, ya que este tiene un orden secuencial.
Presenta una estructura similar al primer tipo, en la que la salida de las capas ocultas alimenta de nuevo a la capa oculta, ayudando así a que la red tenga una noción de lo que sucedió antes.

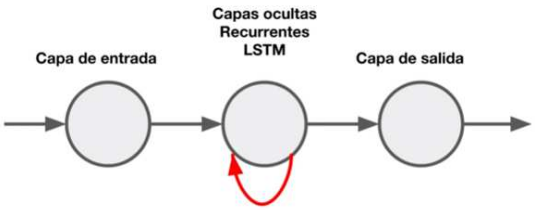

FIGURA A1.3: Representación esquemática del proceso de recurrencia que se da en las redes neuronales recurrentes**.**

Gracias a este modelo han surgido técnicas como el reconocimiento de voz, reconocimiento facial, aplicaciones e interfaces en la Web como recomendación de vídeos en determinadas páginas de la Web o publicidad personalizada según las últimas búsquedas o preferencias, búsqueda de artículos o información mediante palabras clave e interacción mediante la voz con nuestros dispositivos.

En nuestro caso vamos a utilizar un tipo DNN llamado Deep Feedforward Net (o red totalmente conectada o full-conected), que se carcteriza porque la salida de la función de la primera capa es la entrada a la segunda capa, esa salida es la entrada a la tercera capa y así sucesivamente.

A1.1 CAPAS OCULTAS DE LA RED NEURONAL:
Cuanta más cantidad de capas, más matizada puede ser la toma de decisiones, lo cual quiere decir que más conexiones podremos establecer entre las distintas variables o más correlaciones entre ellas, lo que es algo parecido a los términos del desarrollo en serie de Taylor del método estadístico.
Aunque hay casos en los que se ha estudiado cuál es el número óptimo de neuronas o capas que posea la red neuronal, en nuestro caso no lo hay, lo que supondrá una elección de estas a base de prueba y error.

ANEXO 2: ACEPTANCIA DE CAMBIOS O EVOLUCION DE LA CONFIGURACIÓN

Como ya hemos dicho en un apartado anterior, supondremos que nuestro sistema sigue la Mecánica Estadística siguiendo la probabilidad de aparición de una determinada configuración la distribución de Boltzman:

$$p(C_t) \propto e^{-\frac{1}{KT}E(C_t)}$$

Siendo K la constante de Boltzman y C_t la configuración obtenida en un tiempo t, significando t una determinada iteración.

Para la aceptancia o no de la nueva configuración dada por los coeficientes de nuestra función de aproximación utilizamos el método de Metrópolis ya que si suponemos la distribución indicada anteriormente, aunque no conozcamos la probabilidad de una determinada configuración sí que conocemos la relativa o el cociente entre esta y la obtenida por otra configuración cualquiera. El método viene dado por los siguientes pasos:

1. Partimos de una configuración C_t
2. Calculamos su probabilidad relativa $p(C_t)$
3. Calculamos otra configuración aleatoria C_{t+1} y volvemos a calcular su probabilidad relativa $p(C_{t+1})$.
4. Calculamos el cociente de ambas probabilidades: $\gamma = \frac{p(C_{t+1})}{p(C_t)}$
5. Generamos un número aleatorio θ en el intervalo [0, 1] uniformemente.
6. Si $\gamma > \theta$, elegimos la configuración C_{t+1}, si no, nos quedamos con la anterior
7. Volvemos al paso 3 sucesivamente tomando como C_t la elegida en el paso 6.

Vemos que si $p(C_{t+1}) > p(C_t)$, aceptaremos siempre el cambio ya que el cociente será mayor que uno de manera que si la probabilidad de la configuración nueva es mayor aceptaremos el cambio. En caso de que $p(C_{t+1}) < p(C_t)$ también hay cierta probabilidad de aceptar el cambio, dependiendo del cociente entre probabilidades y el número aleatorio generado θ.

Podemos ver que estas probabilidades dependerán del valor de β escogido, de manera que si es muy próximo a cero la exponencial se podrá aproximar a la unidad siendo así casi todas las configuraciones igual de probables y aceptando el cambio en casi todas las iteraciones.

Nosotros no queremos saber la distribución de la energía sino hallar el valor más pequeño de esta, por lo que nuestro algoritmo es algo más simple sin necesidad de guardar nuestras configuraciones sino que solo avanzaremos por estas llegando y guardándonos el punto mínimo alcanzado de la energía que es el equivalente a la probabilidad máxima de determinada configuración.

Cuando hablamos de configuración nueva nos referimos a la configuración dada después de un proceso *sweep* explicado en el apartado 3.3 en el que el cambio en cada índice de los

coeficientes de la funicón aproximada sí que se realiza de forma secuencial, pero los nuevos coeficientes son generados aleatoriamente de forma uniforme en un rango determinado.

ANEXO 3: SELECCIÓN DE TÉRMINOS Y NÚMERO DE ITERACIONES DE LAS SIMULACIONES CON M.E.

Hemos establecido el número de iteraciones y el número de términos de la función aproximación a escoger para la estimación de nuestros datos después de haber realizado varias pruebas. En estas nos hemos basado tanto en los mejores resultados obtenidos como en un tiempo aceptable de compilación de nuestro programa.

Hemos decidido que el número óptimo para nuestro entrenamiento sea un número de 1500 iteraciones y operar solamente con los tres primeros términos de la función de aproximación.

Vemos algunas gráficas que lo ilustren:

Primero hemos entrenado con el error cuadrático medio, llamado “DELTA”, y con este hemos estudiado la influencia en los resultados de la evolución del error con la adicción del término de activación:

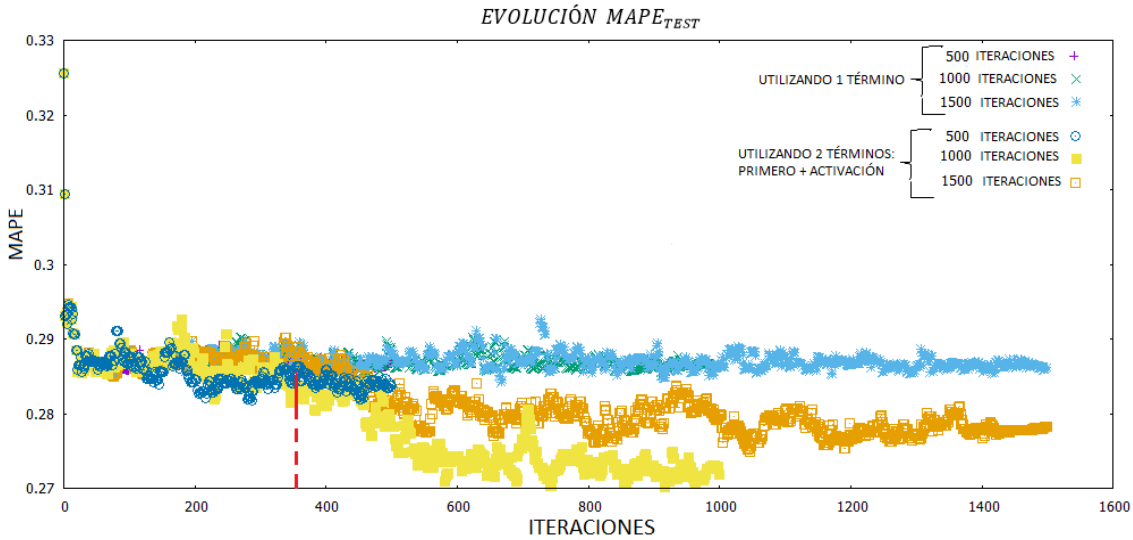

Figura A3.1: Representación de la evolución del error de un mismo modelo escogiendo tres números distintos de iteraciones y divididas en dos grupos. Uno en el que solo utilizamos el primer término de la función de aproximación, con el que ajustamos los distintos pesos de cada variable para la estimación del precio, y un segundo en el que tenemos en cuenta el término de activación, empezando a iterar con este cuando llevaos 350 pasos.

En esta gráfica podemos ver una mejora notable e inmediata al empezar a operar con el segundo término, afirmando así que existe un término de sesgo, estableciendo una función escalón de altura variable en función de un sumatorio pesado de las distintas variables.

Después, hemos realizado pruebas para ver si mejorábamos nuestras predicciones estableciendo como energía de nuestro sistema el error con el que trabajan las empresas y error que dábamos como solución. Esto quiere decir que, ya que damos la solución del MAPE, podemos entrenarlo tomando este error como energía de nuestro sistema, y no el error cuadrático medio con el que trabajamos primeramente:

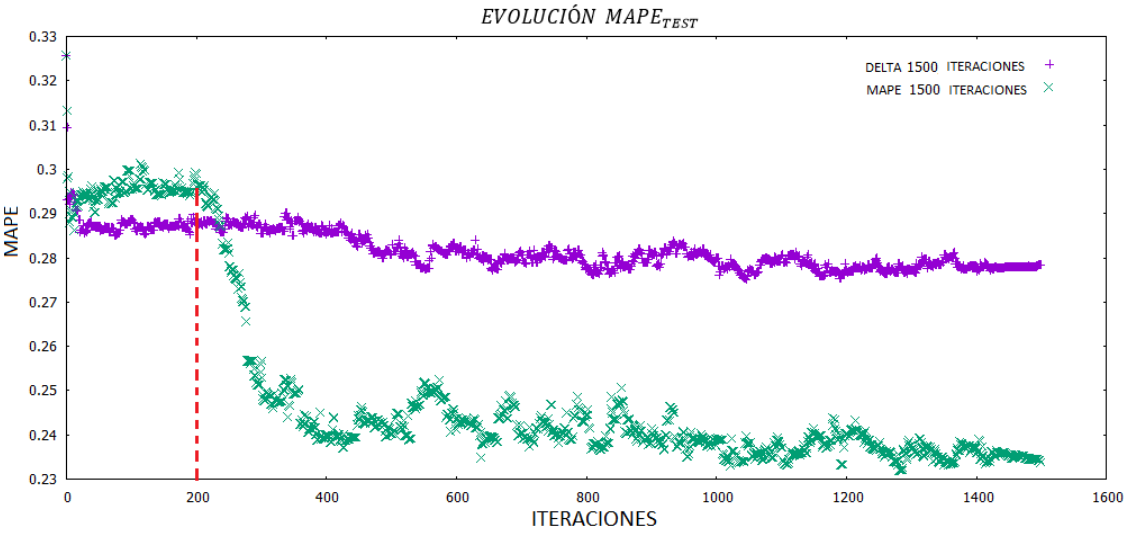

Figura A3.2: Dependencia en la evolución del MAPE de un mismo modelo, tomando como energía el propio MAPE o el error cuadrático medio (RMSE) convencional. En ambas representaciones utilizamos 1500 iteraciones y empezamos a operar con el segundo término cuando llevamos 200 pasos de entrenamiento.

Seguidamente, hemos pasado a estudiar la trascendencia del resto de términos, viendo si añadiendo alguno de estos podemos mejorar nuestros resultados significativamente.

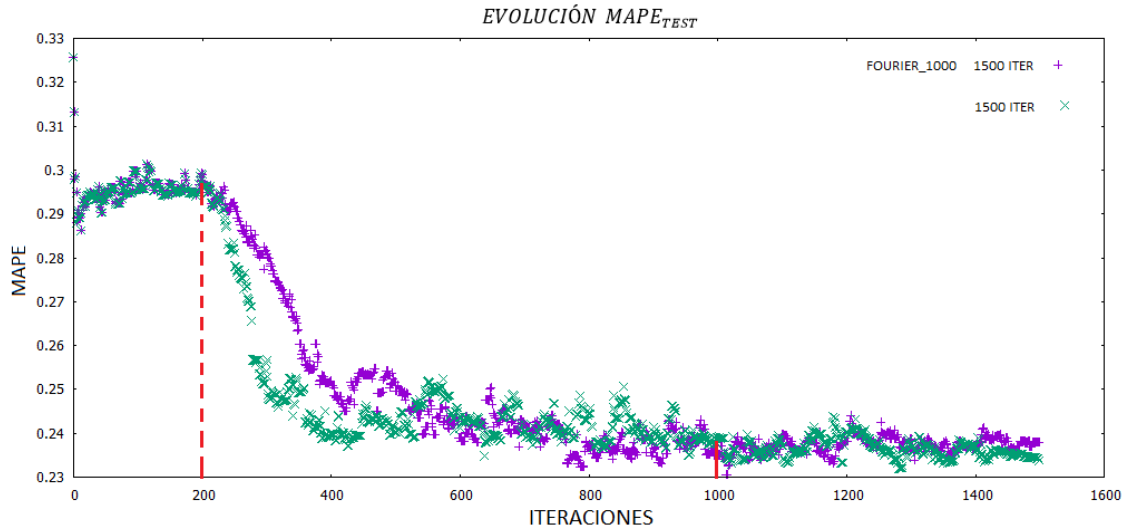

Figura A3.3: Dependencia en la evolución del MAPE de un mismo modelo utilizando en ambas el término de activación a partir de los 200 pasos de entrenamiento y en una de ellas el término de Fourier a partir de los 1000. En ambas representaciones utilizamos 1500 iteraciones.

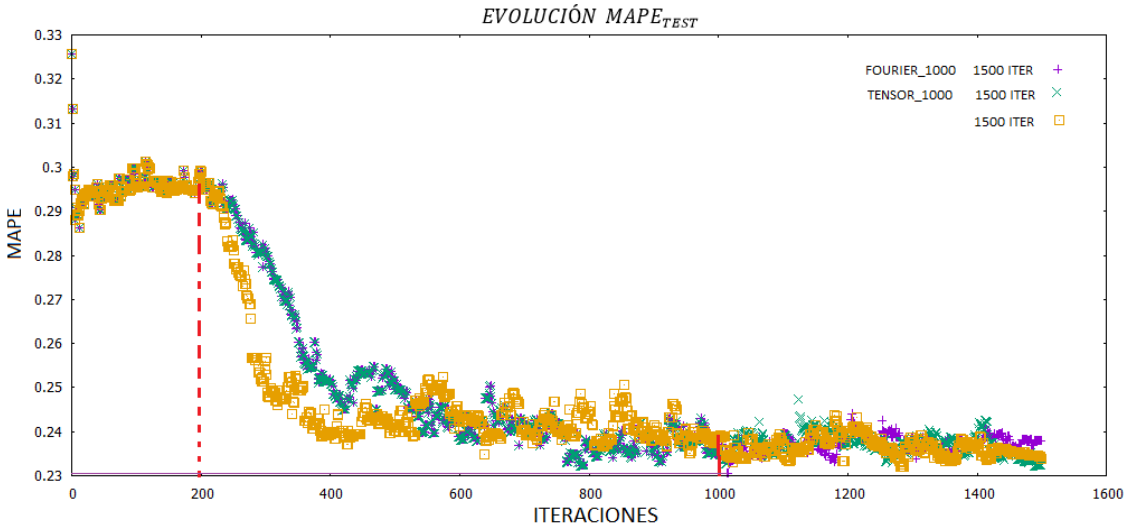

Figura A3.4 : Dependencia en la evolución del MAPE de un mismo modelo utilizando en las tres evoluciones el término de activación a partir de los 200 pasos de entrenamiento, en una de ellas el término de Fourier a partir de los 1000 en otra el término de segundo orden en la expansión de la función en serie de Taylor. Utilizamos 1500 iteraciones en los tres casos.

Aunque las evoluciones son bastante parecidas, sí que podemos apreciar una pequeña mejoría en la obtención de un mínimo si añadimos el término de Fourier o la segunda derivada parcial (tensor que relaciona las variables dos a dos), obteniendo resultados mínimos parecidos.

En nuestro caso hemos decidido operar añadiendo solo el término de Fourier, ya que el tiempo de ejecución del programa se multiplica por 10 si añadimos el término de segundo orden.

Haciendo varias pruebas para modelos de distinto tamaño, hemos obtenido unos resultados similares, por ello hemos establecido el mismo número de términos y de iteraciones para todos ellos.

Finalmente, con estos términos y número de iteraciones escogidos, cada evolución de los distintos modelos ha tardado una media de 10 minutos de ejecución, (variando mucho el tiempo entre esta según el número de datos que contenía cada modelo) y como tenemos 489 modelos distintos, tardando unos 3 días en la obtención de los distintos resultados.

Aunque con el método D.L. la obtención de resultados es más rápida, ya que no guardamos datos intermedios, sino que nos quedamos con el resultado final, al necesitar ver la evolución de los resultados y guardar 50 datos en cada entrenamiento, el tiempo de iteración se multiplicaba por 50, dando una media para el tiempo de ejecución, de algo más de media hora.

Por este motivo en el apartado 7.3 solo se han dado los resultados finales al aumentar el número de iteraciones de entrenamiento para los modelos que no nos dieron un buen resultado.

A3.1 COMPARACION EVOLUCIÓN TENIENDO EN CUENTA MÁS TÉRMINOS DE LA FUNCIÓN

Si hubiéramos dispuesto de más tiempo y nuestro objetivo hubiera sido el de obtener datos mucho más precisos y llegar a ver las distintas relaciones entre las variables de la estimación del precio de un seguro, podríamos haber añadido más términos en la función de aproximación del método de Mecánica Estadística y aumentado el número de iteraciones.

Aunque la influencia de los primeros términos la podíamos ver casi inmediatamente después de operar con ellos, las distintas relaciones entre los valores de las distintas variables tienen cierto tiempo de estabilización. Al principio incluso empeoran nuestros resultados cuando empezamos a entrenar con ellas, pero luego, después de un tiempo empiezan a tomar relevancia en la evolución y empiezan a notarse sus efectos llegando a disminuir el error bastante.

Aquí podemos ver un ejemplo de esto:

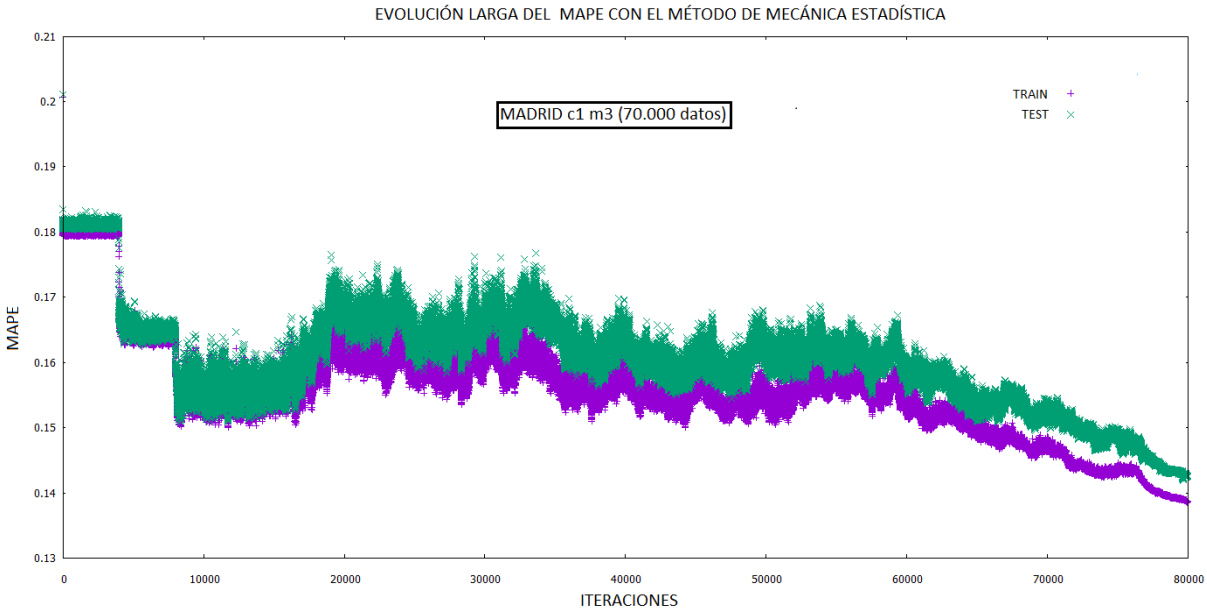

Figura A3.5: Dependencia en la evolución del MAPE tanto de train como de test de un mismo modelo utilizando 80000 pasos de entrenamiento, ejecución que duró algo más de una semana. En esta utilizamos todos los términos de la función de aproximación y podemos ver su efecto claramente. Los términos se han empezado a utilizar en las iteraciones 4000 (activación), 8000 (Fourier), 16000 (Tensor), 24000 (Tensor de activación) y 40000 (Einstein). Vemos que aunque el efecto de la adicción de los dos primeros términos es brusca e inmediata, el de los demás necesita un proceso de estabilización y en cuanto esto sucede empezamos a ver la disminución progresiva del error.

ANEXO 4: LIBRERÍAS Y ARCHIVOS COMPLEMENTARIOS.

A4.1 LIBRERÍAS UTILIZADAS

Además da trabajar con el lenguaje de programación Python, en este método hemos incluido 4 librerías que nos ayudan a manejar nuestros datos:

- TENSORFLOW[12,13] :

Es una librería de código libre para Machine Learning a través de un rango de tareas. Está basado en redes neuronales de aprendizaje profundo y fue desarrollado por Google para satisfacer sus necesidades a partir de redes neuronales artificiales.

- PANDAS[14]:

Es una librería de Python para el análisis de datos que cuenta con las estructuras de datos que necesitamos para limpiar los datos en bruto y que son aptos para el análisis.

Tiene dos estructuras de datos principales:

-Series: Son arrays unidimensionales con indexación (arrays con índice o etiquetados)

-DataFrame: Son estructuras de datos similares a las tablas de bases de datos relacionales como SQL.

Esta librería también incluye funciones para leer datos a partir de diferentes formatos como texto, excell y csv.

- NUMPY[15]:

"Es la librería fundamental para la computación científica con Python, añadiendo soporte para matrices N-dimensionales y una colección de funciones matemáticas de alto nivel para operar con estas matrices"[16].

- OS:

Este módulo proporciona una forma portátil de utilizar la funcionalidad dependiente del sistema operativo e incluye funciones para obtener direcciones de archivos, nombres de archivos y crear carpetas en la dirección que queramos.

A4.2 ARCHIVOS UTILIZADOS

En este método utilizamos cuatro archivos Python cada uno con una función determinada.

Los archivos Python son los siguientes:

- IRIS_DATA.py

Este es un fichero auxiliar, el cual hace una pequeña transformación y especificación de las columnas que contienen nuestros archivos ".csv", para que la librería tensorflow sea capaz de leer bien los datos, y define funciones de evaluación y entrenamiento de nuestros datos.

Leemos los archivos de entrenamiento y testeo (tanto el archivo del precio como el de las demás variables) y, mediante una función de la librería de Pandas, los almacenamos en cuatro variables DataFrame, que son: train_x, train_y, test_x, test_y.

Definimos funciones para la evaluación y entrenamiento de nuestros datos, con las que simplemente dividimos nuestras primas en grupos menores de datos denominados "batch", para que cada vez que estemos entrenando o evaluando nuestros resultados lo hagamos con uno distinto, ya que hacerlo con toda la cantidad de datos de un modelo sería muy costoso. Estos grupos se cambian aleatoriamente en cada iteración de entrenamiento.

En este método vamos entrenando y evaluando con distintos batchs en cada iteración, de distinto modo que en el método anterior, en el que ajustábamos los parámetros de la función en cada iteración teniendo en cuenta todos los datos del archivo de entrenamiento.

Por último, especificamos el nombre de cada columna, que son las distintas características del seguro y especificamos que tipo de variable va a ser toda esa columna dividiéndolas en los tipos vistos en el apartado (2.1).

- LOAD_MOEL.py

En este archivo especificamos el modelo que vamos a utilizar y las variables en tensorflow.

Empezamos especificando qué tipo de variable estamos almacenando en cada columna de nuestro DataFrame con las siguientes funciones, accediendo a cada columna por la variable de su nombre definida en el archivo IRIS_DATA.py:

-tf.feature_column.categorical_column_with_identity: Con esta función especificamos que las distintas columnas a las que se aplica la función van a ser columnas categóricas que devuelven valores de identidad (0 o 1). Esta función se aplica a las distintas covers.

- tf.feature_column.numeric_column: Con esta función especificamos que la columna a la que se le aplica va a representar valores reales o características numéricas. Se aplica al resto de columnas.

Una vez especificado el tipo de variable de todas las columnas, juntamos los nobres de las columnaslas en un vector de tensorflow llamado "my_feature_columns", para unirlo posteriormente a la red neuronal.

Por último, generamos nuestra red neuronal con tensorflow. Esto lo hacemos con la función:

-tf.estimator.DNNRegressor: Esta función nos devuelve un estimados DNNRegressor.

En esta especificamos las capas ocultas que va a tener nuestra red, que en nuestro caso serán cinco, cada una con las siguientes neuronas: [150, 75, 50, 20, 10].

También especificamos las columnas de características que tenemos, que será el vector "my_feature-columns" el cual será la capa de entrada de la red y por ello estará formada por 20 neuronas, una por cada columna.

Además especificamos la función de activación de las neuronas, la cual será la función ReLU.

A4.3 FUNCIÓN DE ACTIVACIÓN[17, 18]:

La neurona es la unidad básica de procesamiento dentro de una red neuronal, la cual tiene unos ciertos valores de entrada con los que la neurona realiza un cálculo interno y genera un valor de salida.

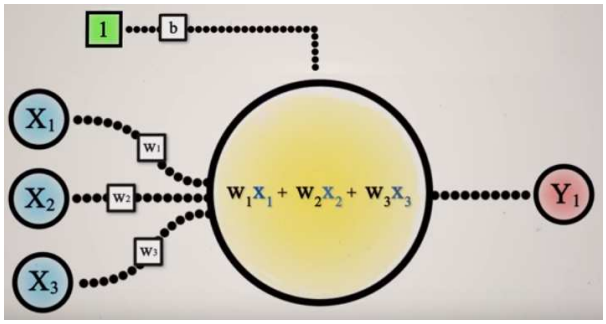

Figura A4.1: Modelo de una neurona en la que vemos sus distintas entradas X_i's con sus respectivos pesos w_i's, que junto con la suma del valor umbral generan la salida Y_i.

Como ya hemos dicho anteriormente, cada neurona de una capa está totalmente conectada a las neuronas de la capa anterior, por ello, esta tendrá por tanto tantas entradas como neuronas tenga la capa anterior.

En el cálculo numérico realizado, la neurona tiene en cuenta todas las entradas que le llegan, haciendo una suma ponderada de esta mediante los pesos que se le asignan a cada una de las entradas. Estos pesos son los parámetros de nuestro modelo y los iremos ajustando en cada iteración para que nuestra red neuronal pueda predecir el valor del precio del seguro.

Además de los pesos de cada entrada solemos tener un parámetro de sesgo, que indica un valor umbral del sumatorio para que la neurona se active, quedando así una función de las distintas entradas:

$$f(\vec{x}, b_j) = \left(\sum_{i=0}^{N_{ENTRADAS}} w_i x_i \right) - b_j$$

Siendo las $x_i's$ las distintas entradas de la neurona, los $w_i's$ los pesos de cada entrada y b_j el sesgo de cada neurona, el cual es distinto para cada una.

La salida de nuestra neurona viene determinada por una *función de activación*, a la que se le pasa el valor de la suma ponderada de nuestras entradas y, dependiendo de ese valor, tendremos un valor de salida u otro.

Generalmente estas funciones de activación son no lineales y devuelven un valor en el rango [0,1] decidiendo de este modo si la neurona está activada o no.

Históricamente, la función sigmoide es la función de activación más antigua y popular, definida como:

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

Esta función ha sido la base de la mayoría de las redes neuronales por muchas décadas, aunque en años recientes han perdido popularidad, debido a que las redes neuronales de muchas capas se vuelven muy difíciles de entrenar, debido al problema de desaparición del gradiente (función que nos sirve para entrenar la red que explicaremos más adelante).

En su lugar, la mayoría de redes neuronales utilizan otro tipo de funciones de activación, como es en nuestro caso, que trabajamos con la función de activación ReLU[19] (Rectified Linear Unit):

$$R\big(z = f(\vec{x}, b_j)\big) = \max(0, z))$$

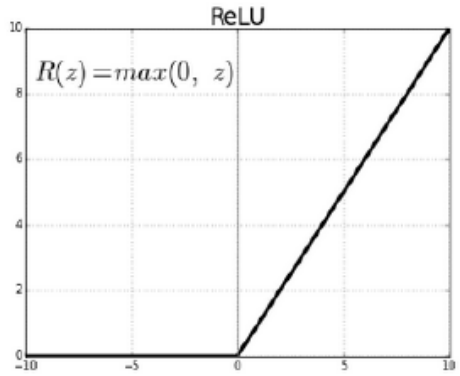

Figura A4.2 :Función de activación ReLU

Con esta función obtenemos un valor de la salida si la suma ponderada de las entradas es menor que cero, simulando que la neurona está desactivada, y el mismo valor de la suma si este es mayor que cero, simulando que la neurona está activada y enviando una respuesta a las neuronas de la siguiente capa.

ANEXO 5: FUNCIÓN DE COSTE

Para entrenar nuestro modelo definimos una *función de coste*[20], la cual evalúa el error que tenemos de nuestras predicciones y de los valores reales obtenidos de nuestro precio, siendo esta generalmente el error cuadrático medio entre salidas de la red neuronal y valores reales:

$$F_{COSTE} = \frac{1}{N_{datos}} \sum_{i=0}^{N_{datos}} (Y_{PREDICCIÓN} - Y_{REAL})^2$$

Un optimizador es un operador que, dada la función de coste, hace un reajuste de nuestros parámetros minimizando el valor de esta función mediante el algoritmo denominado "Backpropagation" y así, entrenando nuestro modelo.

Este algoritmo se basa en el método del descenso del gradiente[21], vector que nos indica la dirección hacia la que nuestra pendiente asciende. Por tanto, como queremos encontrar el mínimo de nuestra función, deberemos evolucionar en sentido opuesto a este vector, "caminando cuesta abajo". Otro parámetro que debemos tener en cuenta es el ratio de aprendizaje, que indica cuánto afecta el gradiente a la actualización de nuestros parámetros en cada iteración, aspecto que veremos más adelante.

Como la función de coste depende de la salida de nuestra red, que a su vez depende de los distintos pesos de las distintas salidas de todas las neuronas de la última capa oculta, que a su vez dependen de los distintos pesos de las distintas salidas de las neuronas de la capa anterior y así sucesivamente, el gradiente de la función se calculará mediante derivadas parciales utilizando la *regla de la cadena*, de ahí su nombre de "Backpropagation"[22].

Cada derivada parcial tendrá un valor, el cual dictará la importancia de este en la variación de la función de coste, viendo así que neuronas y capas son las más importantes para la variación de la función.

ANEXO 6: COMPARACIÓN DE RESULTADOS OBTENIDOS CON OTROS PROPORCIONADOS POR ALFREDO

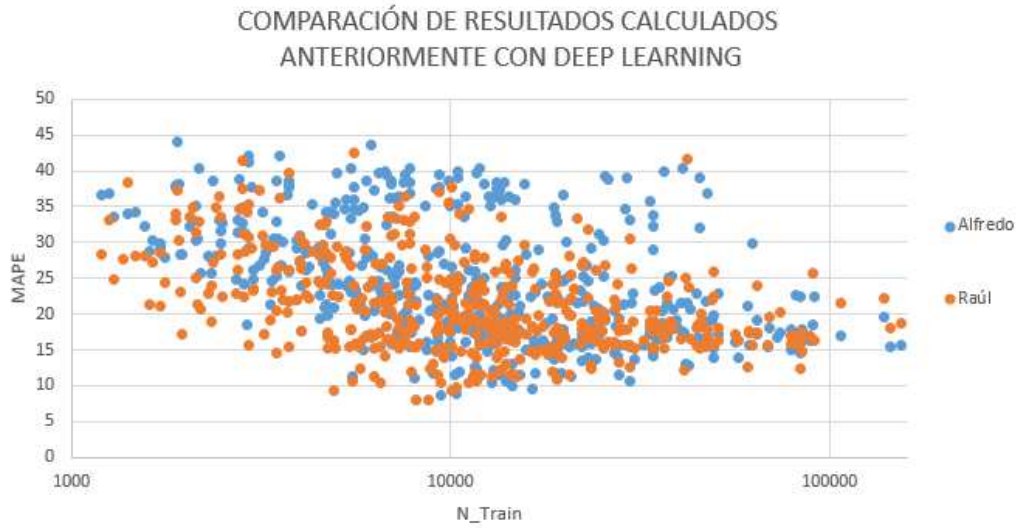

Figura A6: Comparación de nuestros resultados con otros calculados anteriormente y proporcionados por el ayudante Alfredo.

Vemos que los datos son parecidos a los obtenidos por Alfredo anteriormente, comprobando así que el problema no está en el algoritmo empleado.

Realizando la media ponderada de estos resultados obtenemos los siguientes valores:

$$\langle MAPE \rangle_{ALFREDO} = 22.09\% \quad \langle MAPE \rangle_{RAÚL} = 18.32\%$$

Obteniendo un error relativo entre ellos del:

$$ERROR_{RELATIVO} = 20.6\%$$

En nuestro caso hemos obtenido un valor del $\langle MAPE \rangle$ más pequeño, ya que realizamos más iteraciones y entrenamos con un dato más, ya que al principio, el primer dato de cada modelo nos lo saltábamos y no trabajamos con él.

Debido a que el número de pasos de entrenamiento fue cinco veces el utilizado por Alfredo y obtuvimos unos resultados similares, supusimos que el número de estos estaba bien determinado, que el error habría llegado a un mínimo y que la adicción de más pasos no modificaría mucho la mejora de nuestros datos.

ANEXO 7: COMPARACIÓN DE EVOLUCIONES ENTRE AMBOS MÉTODOS

A7.1 FIGURAS QUE MUESTRAN LAS DIFERENCIAS EN LA EVOLUCIÓN DE DISTINTOS MODELOS COMPARANDO LOS DOS MÉTODOS.

MODELOS GRANDES

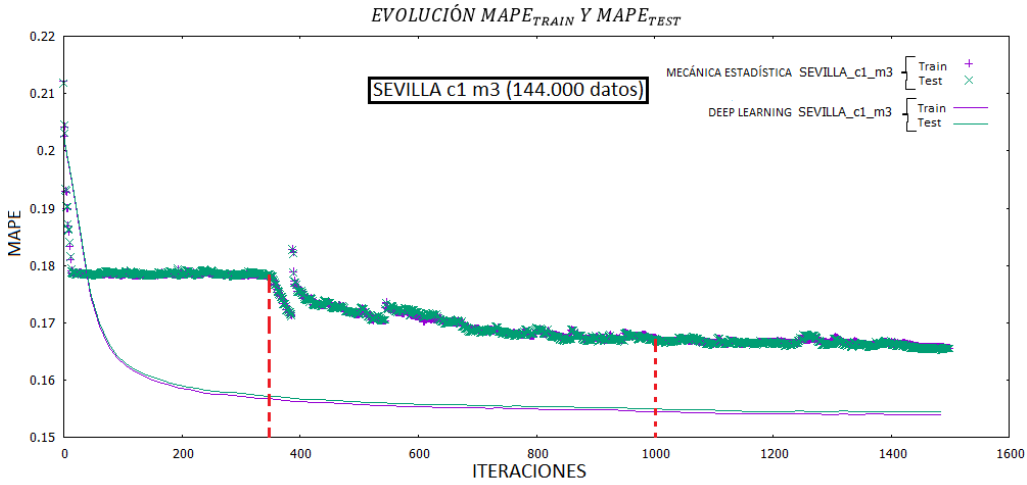

Figura A7.1: Representación de la evolución del MAPE del modelo Sevilla c1 m3,(proporcionado por la compañía 1 y de la modalidad3), con 144.000 datos de entrenamiento.

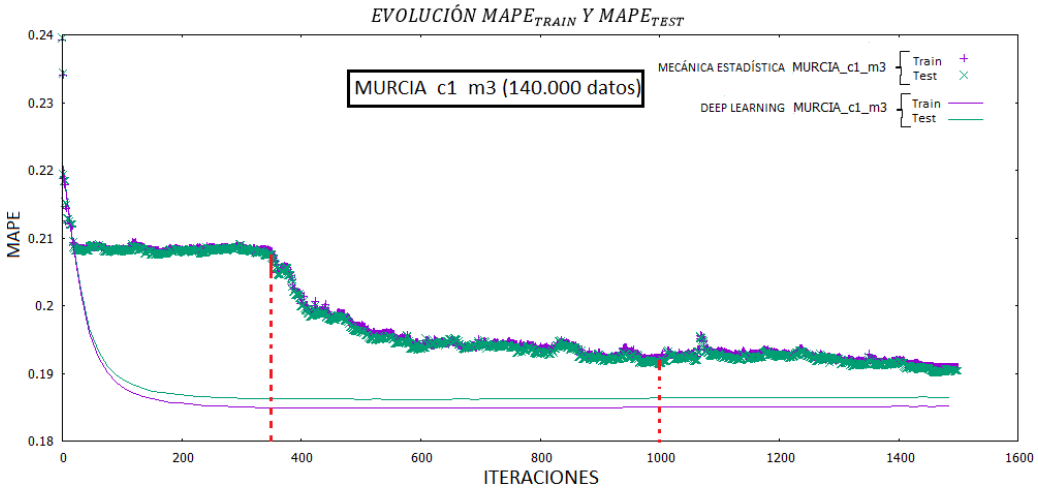

Figura A7.2: Representación de la evolución del modelo Murcia c1 m3 con 140.000 datos de entrenamiento.

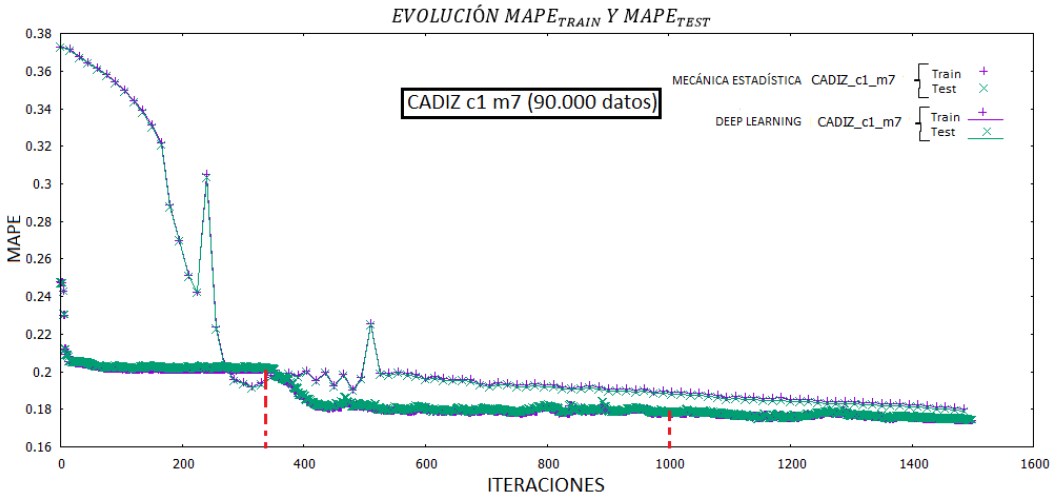

Figura A7.3: Representación de la evolución del modelo Murcia c1 m7 con 90.000 datos de entrenamiento.

MODELOS MEDIOS

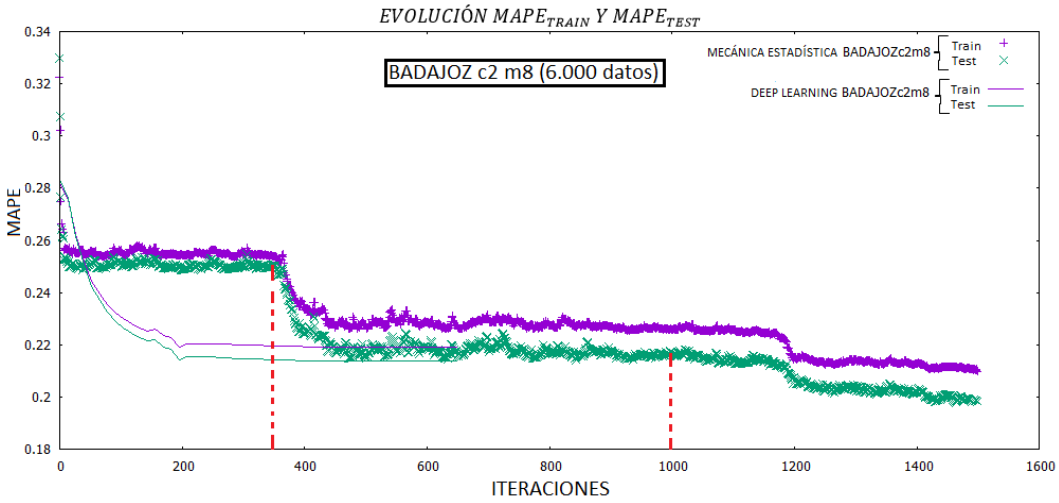

Figura A7.4: Representación de la evolución del modelo Badajoz c2 m8 con 6.000 datos de entrenamiento.

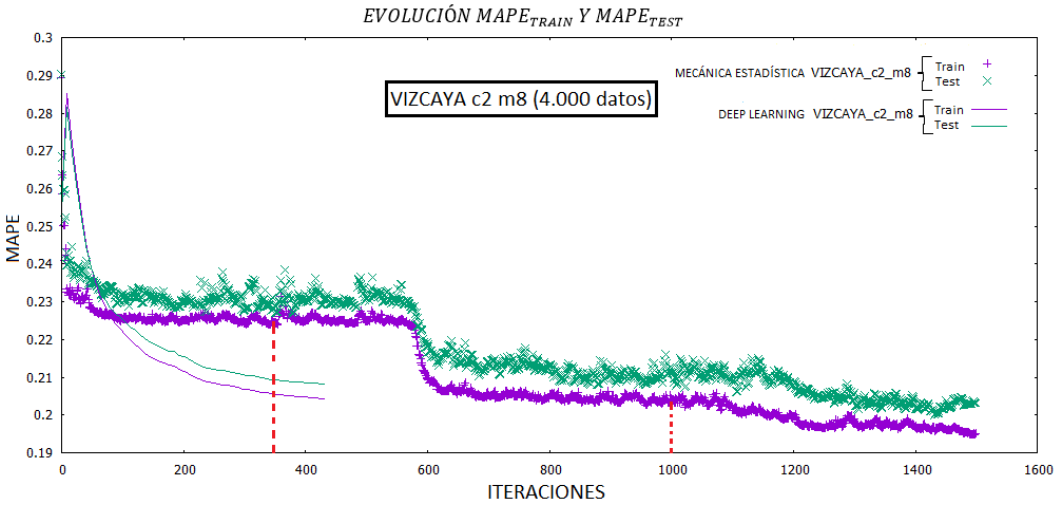

Figura A7.5: Representación de la evolución del modelo Vizcaya c2 m8 con 4.000 datos de entrenamiento.

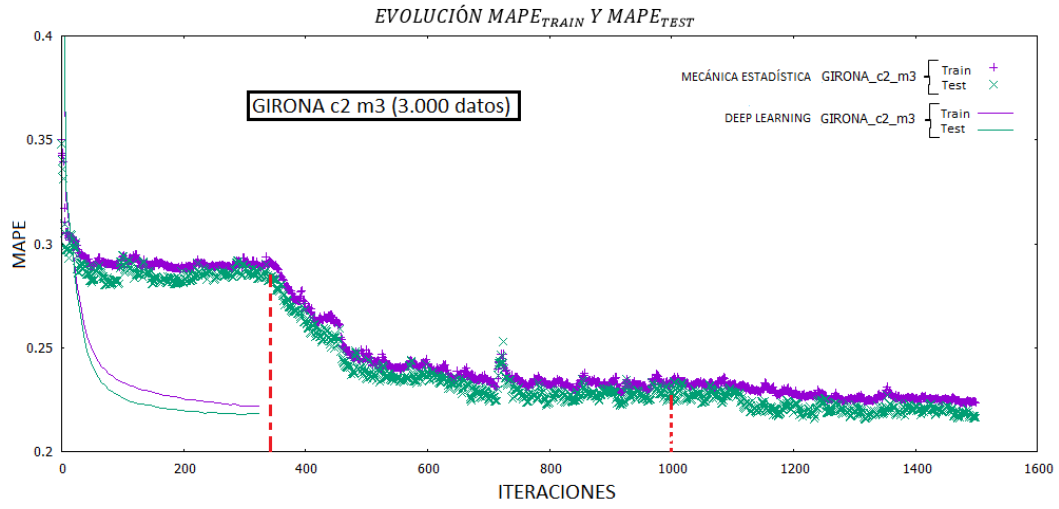

Figura A7.6: Representación de la evolución del modelo Girona c2 m3 con 3.000 datos de entrenamiento.

MODELOS PEQUEÑOS

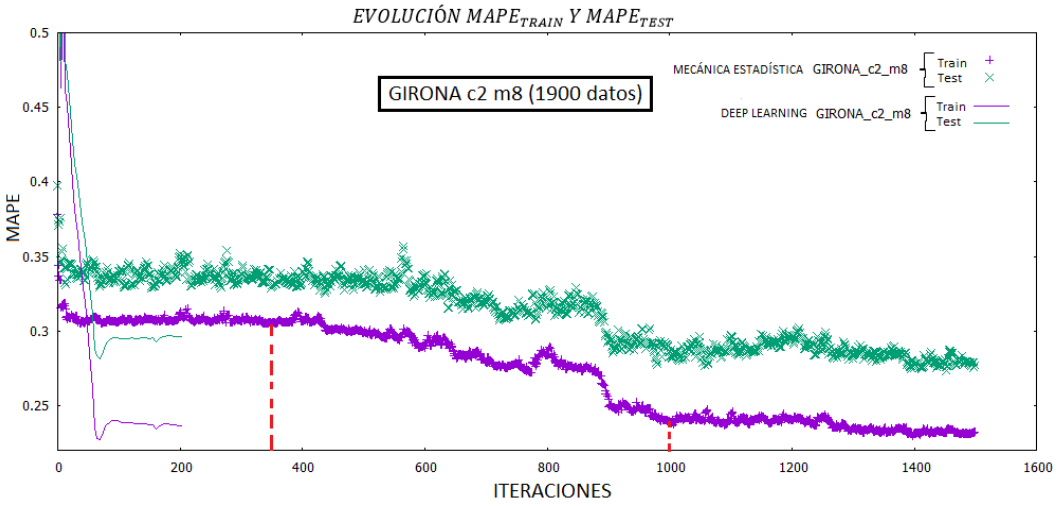

Figura A7.7: Representación de la evolución del modelo Girona c2 m8 con 1900 datos de entrenamiento.

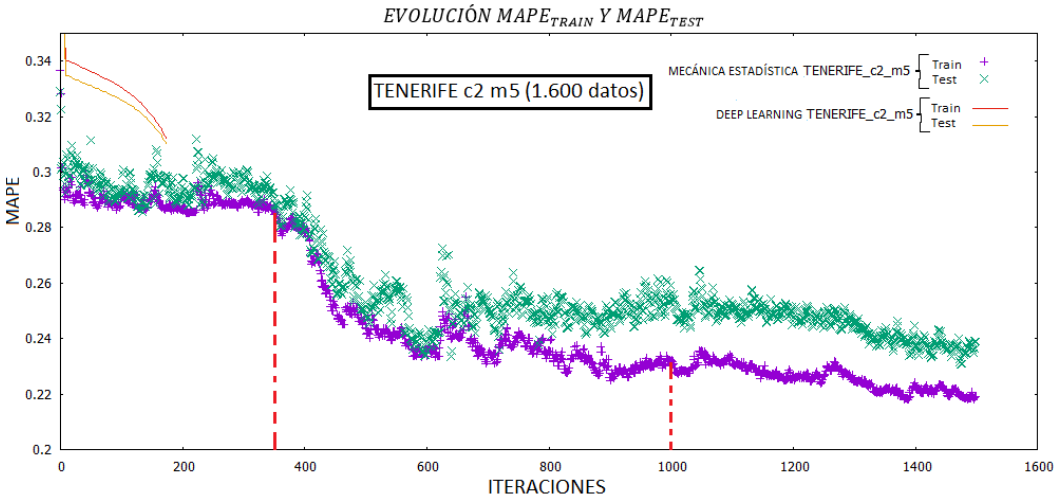

Figura A7.8: Representación de la evolución del modelo Tenerife c2 m5 con 1600 datos de entrenamiento.

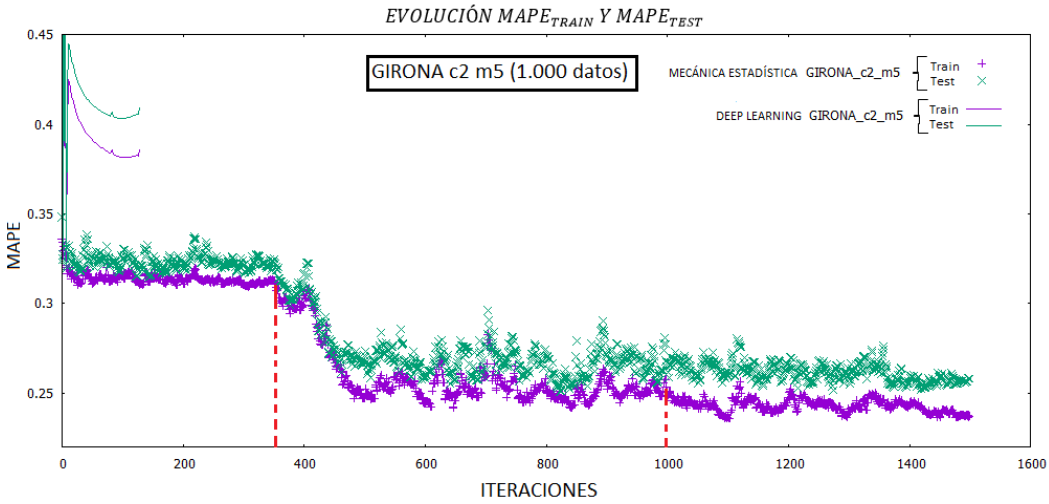

Figura A7.8: Representación de la evolución del modelo Girona c2 m5 con 1000 datos de entrenamiento.

A7.2 FIGURAS QUE MUESTRAN LA EVOLUCIÓN DEL MAPE AUMENTANDO EL NÚMERO DE ITERACIONES DE ENTRANEMIENTO.

MODELOS MEDIOS

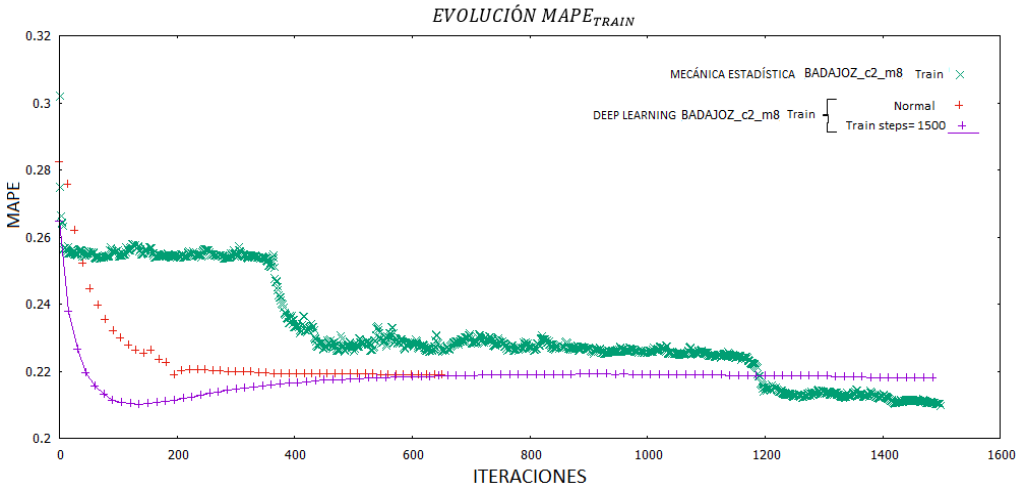

Figura A7.9.1: Representación de la evolución del MAPE Train del modelo Badajoz c2 m8 con el método D.L. tomando 1500 iterciones de entrenamiento.

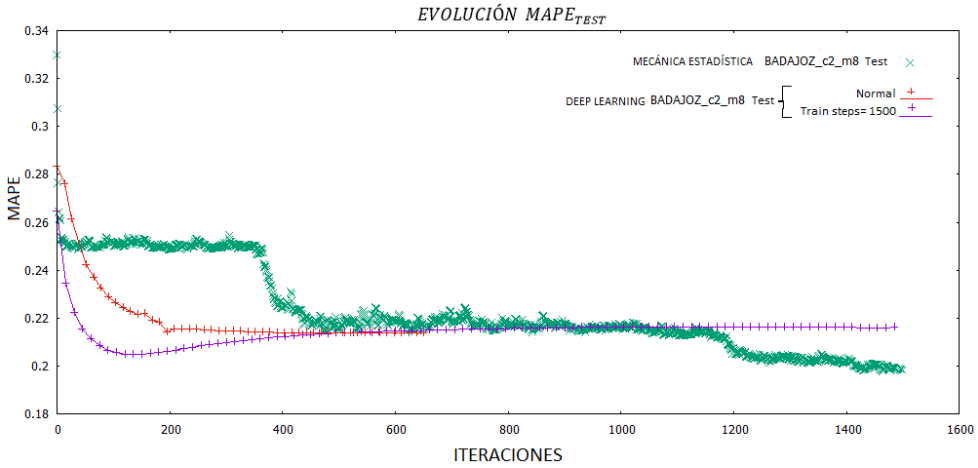

Figura A7.9.2: Representación de la evolución del MAPE Test del modelo Badajoz c2 m8 con el método D.L. tomando 1500 iterciones de entrenamiento.

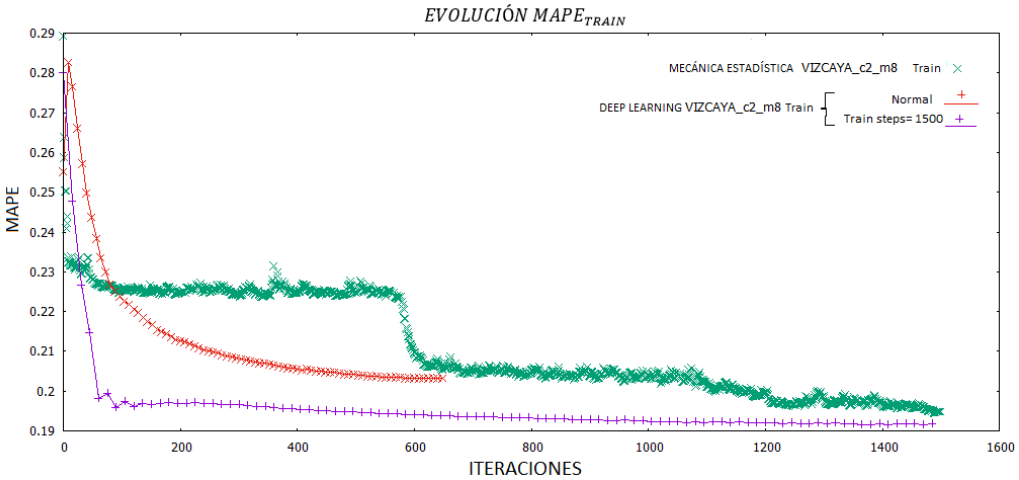

Figura A7.10.1: Representación de la evolución del MAPE Train del modelo Vizcaya c2 m8 con el método D.L. tomando 1500 iterciones de entrenamiento.

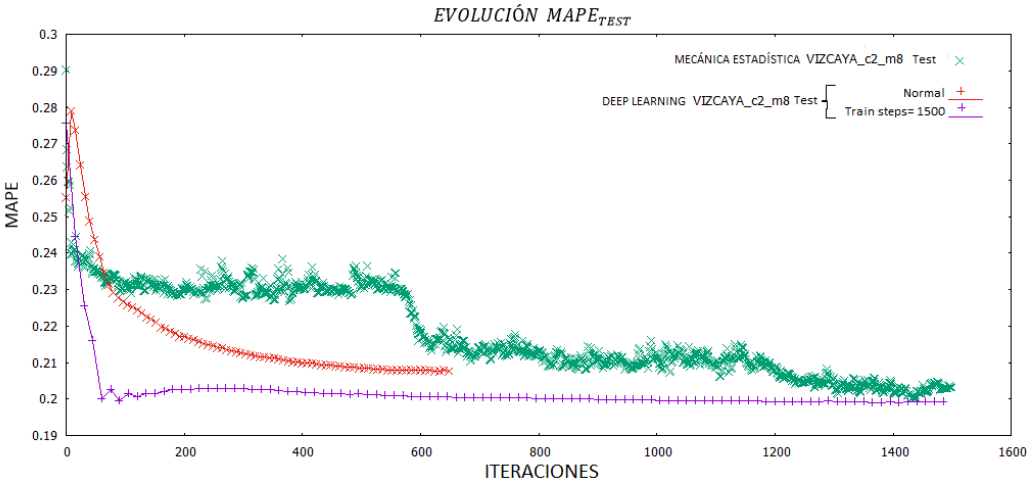

Figura A7.10.2: Representación de la evolución del MAPE Test del modelo Vizcaya c2 m8 con el método D.L. tomando 1500 iterciones de entrenamiento.

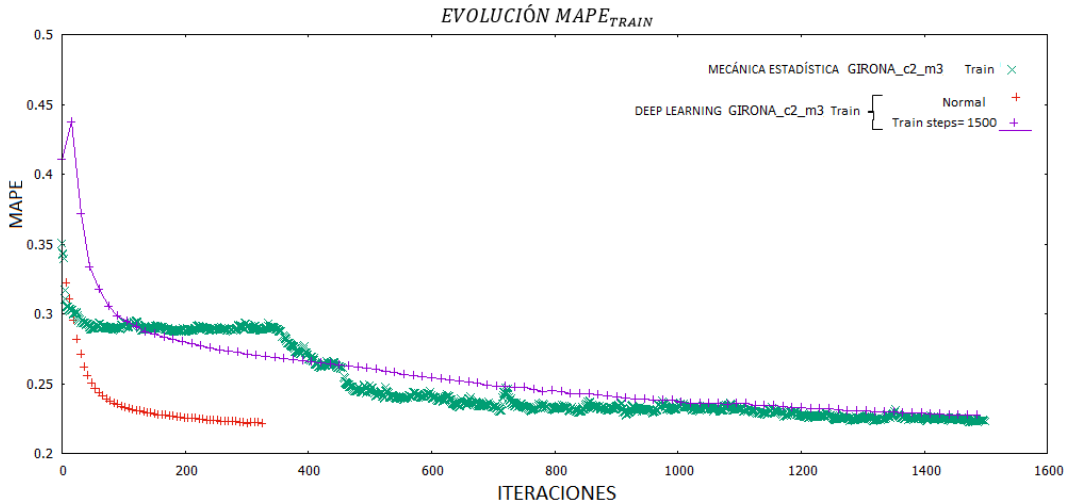

Figura A7.11.1: Representación de la evolución del MAPE Train del modelo Girona c2 m3 con el método D.L. tomando 1500 iterciones de entrenamiento.

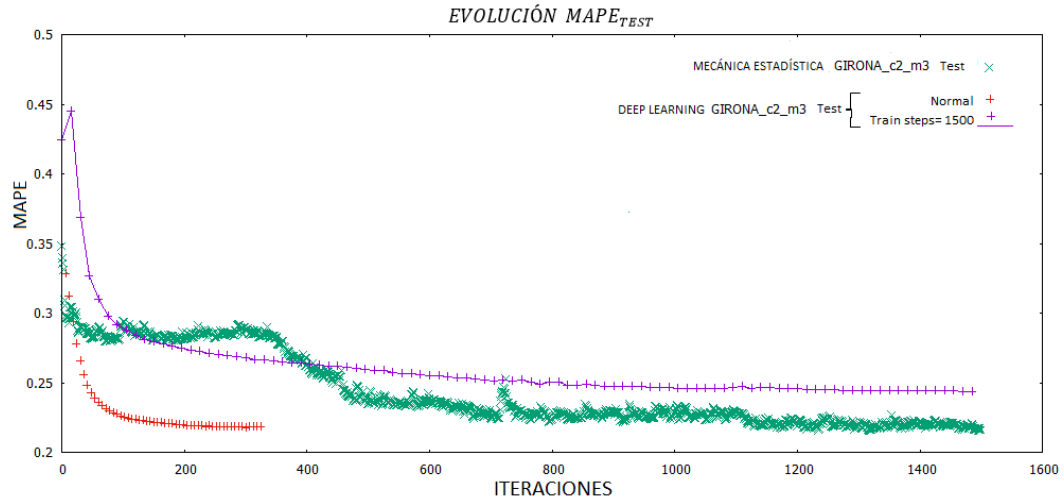

Figura A7.11.2: Representación de la evolución del MAPE Test del modelo Girona c2 m3 con el método D.L. tomando 1500 iterciones de entrenamiento.

MODELOS PEQUEÑOS

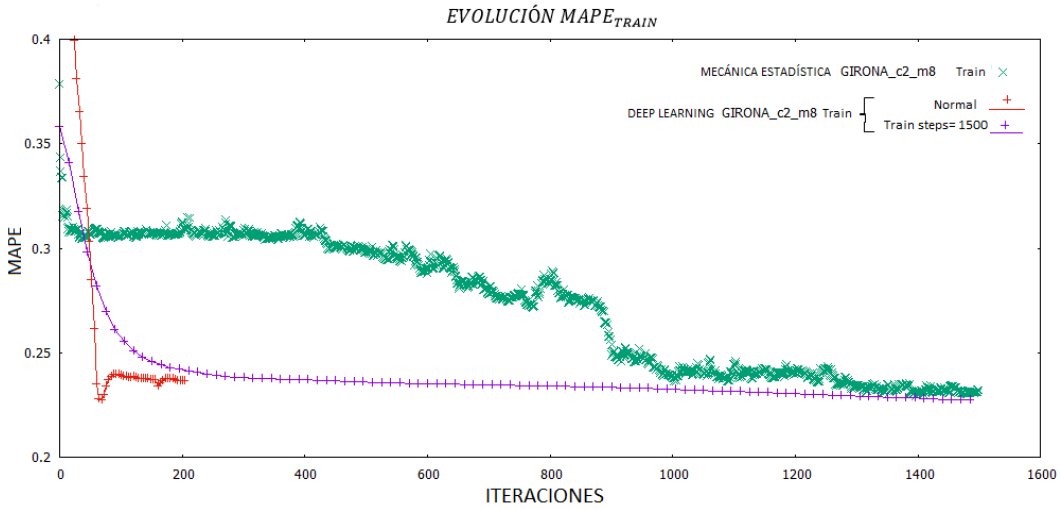

Figura A7.12.1: Representación de la evolución del MAPE Train del modelo Girona c2 m8 con el método D.L. tomando 1500 iterciones de entrenamiento.

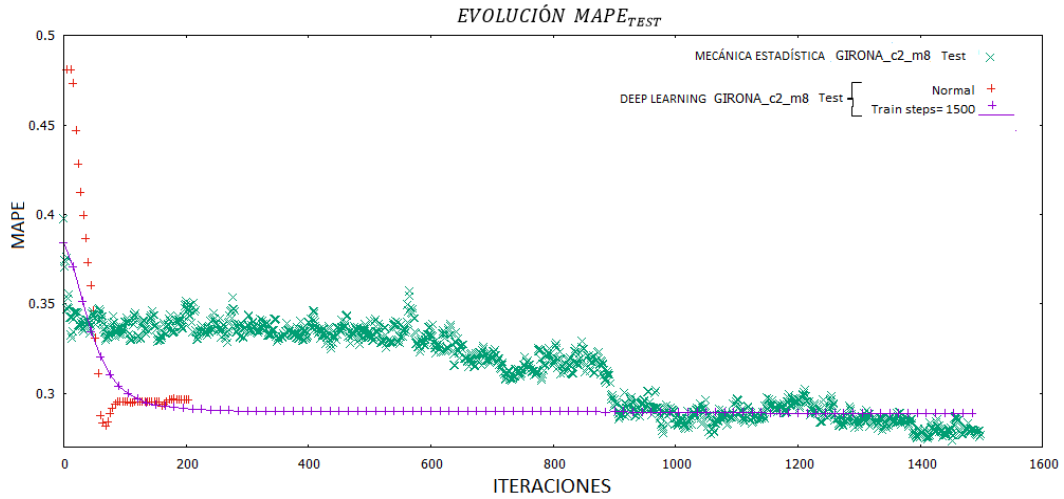

Figura A7.12.2: Representación de la evolución del MAPE Test del modelo Girona c2 m8 con el método D.L. tomando 1500 iterciones de entrenamiento.

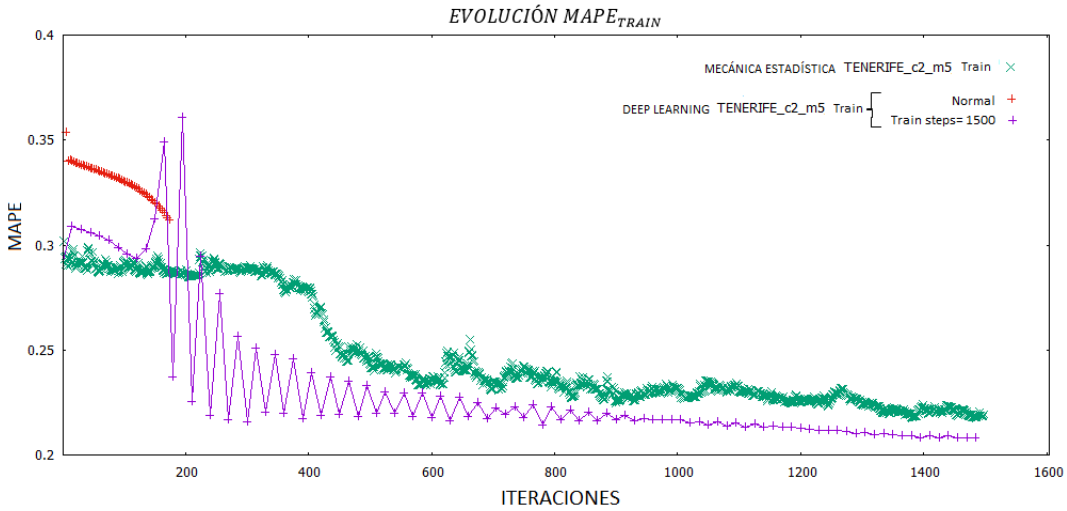

Figura A7.13.1: Representación de la evolución del MAPE Train del modelo Tenerife c2 m5 con el método D.L. tomando 1500 iterciones de entrenamiento.

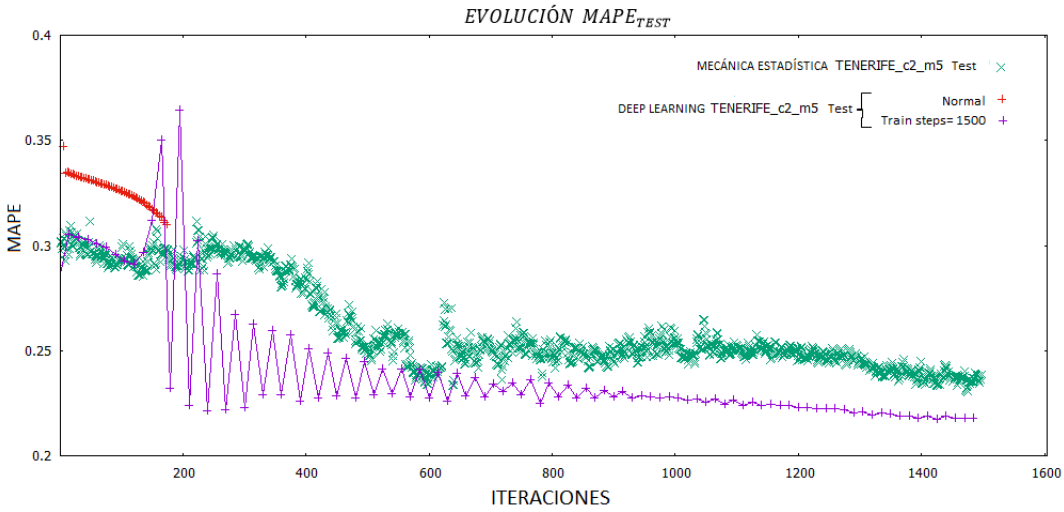

Figura A7.13.2: Representación de la evolución del MAPE Test del modelo Tenerife c2 m5 con el método D.L. tomando 1500 iterciones de entrenamiento.

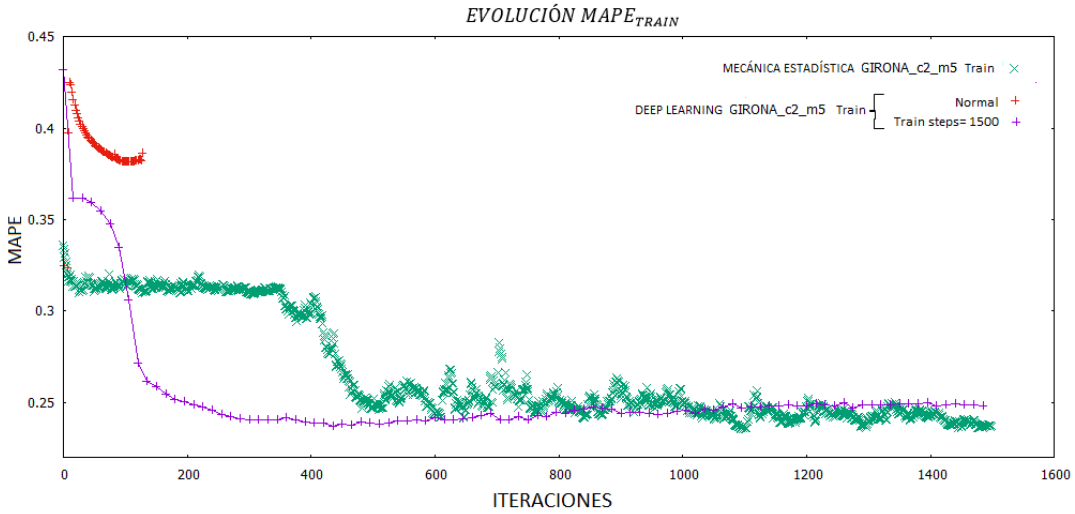

Figura A7.14.1: Representación de la evolución del MAPE Train del modelo Girona c2 m5 con el método D.L. tomando 1500 iterciones de entrenamiento.

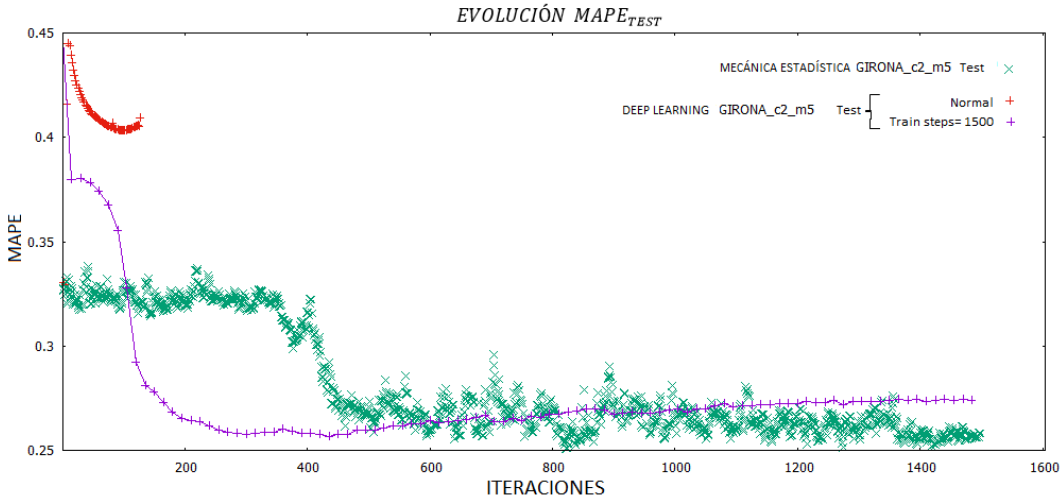

Figura A7.14.2: Representación de la evolución del MAPE Test del modelo Girona c2 m5 con el método D.L. tomando 1500 iterciones de entrenamiento.