Trabajo de Final de Master

Master Data Science

Predicción de la demanda eléctrica de puntos de recarga de vehículos eléctricos mediante Deep Learning

Autor: David Piqué Prat

Julio 2024

https://github.com/davidpique11/DemandForecastEV.git

ÍNDICE

1 Introducción 1

1.1 Motivación 1

1.2 Objetivos 1

2 Herramientas utilizadas 3

3 Descripción del Dataset 4

3.1 Obtención del Dataset 4

3.2 Análisis Exploratorio de Datos (EDA) 5

4 Preprocessamiento de los datos (Feature Engineering) 6

4.1 Feature Engineering 6

4.2 Visualización de los datos 7

5 Metodología 10

5.1 Estrategia y flujo de trabajo 10

5.2 Modelos 11

5.2.1 Red neuronal recurrente LSTM (LSTM) 12

5.2.2 Red neuronal recurrente LSTM doble capa 13

5.2.3 Red neuronal recurrente LSTM triple capa 14

5.2.4 Red neuronal recurrente LSTM multicapa 15

5.2.5 Red neuronal recurrente LSTM multicapa alternada 16

5.2.6 Red neuronal recurrente LSTM bidireccional multicapa 17

5.2.7 Red neuronal recurrente LSTM bidireccional multicapa alternada 18

5.3 Entrenamiento de los modelos 19

5.3.1 Normalización de los datos 19

5.3.2 División de los datos en entrenamiento, validación y test 20

5.3.3 Creación de los generadores secuenciales 21

5.3.4 Carga y compilación del modelo 22

5.3.5 Regularizaciones (Callbacks) 22

5.3.6 Entrenamiento del modelo 23

5.4 Validación y afinamiento del modelo 24

5.5 Test 25

5.5.1 Predicción sobre el conjunto de entrenamiento y de test 26

5.5.2 Predicción a futuro 27

6 Resultados 31

6.1 Selección del nivel de agregación 31

6.2 Entrenamiento 32

6.3 Validación 35

6.4 Test 36

7 Conclusiones 41

8 Referencias 42

Índice de Códigos

Código 4.1: Cálculo de la potencia media de cada sesión 6
Código 4.2: Cálculo de la demanda agregada y creación del nuevo dataframe 7
Código 5.1: Función get_model para seleccionar el modelo 12
Código 5.2: Función para construir el modelo de red neuronal LSTM 12
Código 5.3: Función para construir el modelo de red neuronal LSTM doble capa 13
Código 5.4: Función para construir el modelo de red neuronal LSTM triple capa 14
Código 5.5: Función para construir el modelo de red neuronal LSTM multicapa 15
Código 5.6: Función para construir el modelo de red neuronal LSTM multicapa alternada 16
Código 5.7: Función para construir el modelo de red neuronal LSTM bidireccional multicapa 17
Código 5.8: Función para construir el modelo de red neuronal LSTM bidireccional multicapa alternado 18
Código 5.9: Función que carga los datos según el nivel de agregación 19
Código 5.10: Normalización de los datos 20
Código 5.11 Separación del conjunto de datos en train, validation y test. 20
Código 5.12: Creación de los generadores secuenciales 21
Código 5.13: Proceso de carga del modelo 22
Código 5.14: Compilación del modelo 22
Código 5.15: Configuración de los callbacks 23
Código 5.16: Entrenamiento del modelo 24
Código 5.17: Fase de validación y recopilación de las métricas 25
Código 5.18: Representación de las predicciones 26
Código 5.19: Función para predecir más de un valor en el futuro y calcular el error relativo 28
Código 5.20: Implementación de la función en bucle para 7 días consecutivos 29
Código 5.21: Bucle para comparar los distintos modelos 30

Índice de Figuras

Figura 4.1: Representación de la demanda agregada diaria 8
Figura 4.2: Representación de la demanda agregada horaria 8
Figura 4.3: Representación del pico de demanda puntual 9
Figura 5.1: Diagrama del flujo de trabajo 11
Figura 5.2: Esquema del modelo LSTM 13
Figura 5.3: Esquema del modelo LSTM doble capa 13
Figura 5.4: Esquema del modelo LSTM triple capa 14
Figura 5.5: Esquema del modelo LSTM multicapa 15
Figura 5.6: Esquema del modelo LSTM multicapa alternada 16
Figura 5.7: Esquema del modelo LSTM bidireccional multicapa 17
Figura 5.8: Esquema del modelo LSTM bidireccional multicapa 18
Figura 6.1: Historial de entrenamiento del modelo LSTM (ws = 96, bs = 32) 33
Figura 6.2: Historial de entrenamiento del modelo LSTM_stacked (ws = 96, bs = 32) 33
Figura 6.3: Historial de entrenamiento del modelo LSTM_stacked_2 (ws = 96, bs = 32) 34
Figura 6.4: Historial de entrenamiento del modelo LSTM_stacked_3 (ws = 96, bs = 32) 34
Figura 6.5: Historial de entrenamiento del modelo LSTM_stacked_3_alt (ws = 96, bs = 32) 34
Figura 6.6: Historial de entrenamiento del modelo Bidirectional_LSTM (ws = 96, bs = 32) 35
Figura 6.7: Historial de entrenamiento del modelo Bidirectional_LSTM_alt (ws = 96, bs = 32) 35
Figura 6.8: Fragmento de las predicciones del conjunto de entrenamiento del modelo LSTM_stacked_3 36
Figura 6.9: Fragmento de las predicciones del conjunto de test del modelo LSTM_stacked_3 37
Figura 6.10: Predicción a futuro de 24h con un horizonte de 24h 38
Figura 6.11: Predicciones a futuro de 7 días con un horizonte de 24h 39
Figura 6.12: Comparación de las predicciones de los modelos en horizontes de 24h 40

Índice de Tablas

Tabla 6.1: Comparación de los niveles de agregación diarios y horarios 31
Tabla 6.2: Resultados de los errores absolutos y del MAPE 37

1 Introducción

En la actualidad, la transición energética sostenible y respetuosa con el medio ambiente ha cobrado una gran importancia dentro de la sociedad. Dentro de este marco, una de las principales tendencias es el incremento de la movilidad eléctrica, que ofrece una solución viable para reducir las emisiones de gases contaminantes y la dependencia de los combustibles fósiles. En el caso de los vehículos eléctricos (VE), su adopción se ha visto incrementada en los últimos años, por lo que se ha generado la urgente necesidad de infraestructuras adecuadas, incluyendo la implantación de puntos de recarga eficientes y accesibles que puedan satisfacer la creciente demanda eléctrica [1].

Este trabajo de Fin de Máster (TFM) se centra la predicción de la demanda eléctrica en puntos de recarga de VE mediante la aplicación y desarrollo de técnicas avanzadas de Deep Learning, más concretamente, mediante el uso de Redes Neuronales de Memoria a Largo Corto Plazo (Long Short-Term Memory, LSTM). Las LSTM son una clase especial de redes neuronales recurrentes (RNN) creadas para aprender y recordar patrones en secuencias de datos a largo plazo [2], lo que las hace particularmente adecuadas para análisis de series temporales como la predicción de la demanda eléctrica.

1.1 Motivación

La previsión de la demanda de electricidad en los puntos de recarga tiene importantes implicaciones tanto técnicas como económicas. Una previsión adecuada permite a los operadores de la red y a los proveedores de energía optimizar la gestión de los recursos, las mejoras de las infraestructuras y la planificación del mantenimiento, a la vez que les proporciona las herramientas necesarias para crear estrategias de tarificación más efectivas También facilita la integración de fuentes de energía renovables, como la solar y la eólica, que son intermitentes, lo que se traduce en una mayor estabilidad y sostenibilidad de la red eléctrica.

1.2 Objetivos

El principal objetivo de este TFM es diseñar, implementar y evaluar un modelo de red neuronal LSTM para la predicción de la demanda eléctrica en puntos de recarga de VE que pueda ser posteriormente utilizado como un Software as a Service (SaaS) por los operadores de las estaciones de recarga. Para realizar este objetivo, se detallan los siguientes puntos:

1. Adquisición, preprocesamiento y análisis exploratorio de los datos de consumo eléctrico en varios puntos de recarga de VE.

2. Diseño y entrenamiento de diferentes modelos LSTM, incluyendo la selección de hiperparámetros óptimos y la implementación de técnicas de regularización para evitar el sobreajuste (overfitting).

3. Evaluación del rendimiento de los modelos propuestos mediante métricas como el Error Absoluto Medio (MAE), el Error Cuadrático medio (RMSE) y el Error Porcentual Absoluto Medio (MAPE).

4. Análisis y representación de los resultados obtenidos.

2 Herramientas utilizadas

Para realizar este proyecto, diversas herramientas técnicas se han utilizado:

- **Visual Studio Code**: es un editor de código fuente desarrollado por Microsoft, altamente personalizable y que soporta una gran variedad de lenguajes de programación y extensiones como, por ejemplo, Jupyter Notebook

- **Jupyter Notebook**: en este caso integrado en Visual Studio Code, es una aplicación web de código abierto que permite crear y analizar documentos que contienen código en vivo, visualizaciones y texto. La mayoría del trabajo se realizó utilizando esta herramienta ya que permite visualizar resultados de manera inmediata y documentar todo el proceso de análisis de datos de forma sencilla.

- **Python**: es un lenguaje de programación de propósito general y popularizado por su sintaxis sencilla y su gran compatibilidad con el mundo de la ciencia de datos y la inteligencia artificial. Se utilizaron las siguientes librerías:

    - Análisis de datos y manipulación: numpy, pandas
    - Visualización: matplotlib, seaborn
    - Machine learning: sckit-learn, tensorflow, keras

    Se utilizó la versión python 3.12.0

3 Descripción del Dataset

Para llevar a cabo un proyecto de manera efectiva, es realmente esencial contar con un conjunto de datos adecuado, lo que implica una cantidad suficiente de muestras, calidad en los valores y riqueza en el número de variables. Por este motivo, se buscó un conjunto de datos que cumpliera con estos criterios.

3.1 Obtención del Dataset

El dataset utilizado en este proyecto proviene de un artículo científico [3] publicado el marzo de 2024 por Keon Baek, Eunjung Lee y Jinho Kim y que contiene las sesiones de recarga de vehículos eléctricos registradas por una empresa operadora durante un año comercial en Corea del Sur. El dataset es constituido por 72,856 sesiones de 2,337 usuarios y 2,119 puntos de recarga.

En la siguiente tabla se ven detalladas todos los atributos existentes:

Column	Type	Description
UserID	int64	User ID; own members (1–2337) and other company's members/non-members (0)
ChargerID	int64	Charger ID
ChargerCompany	int64	Categorization by charger company's type: own company (1), other company (0)
Location	object	Installed location of charger; location type (accommodation, apartment, bus garage, camping, company, golf, hotel, market, public area, public institution, public parking lot, resort, restaurant, and sightseeing)
ChargerType	int64	Categorization by charging speed; fast charger (1) and slow charger (0)
StartDay	object	Start date of connection between EV and charger (YYYY-MM-DD)
StartTime	object	Start time of connection between EV and charger (HH:MM:SS)
EndDay	object	End date of connection between EV and charger (YYYY-MM-DD)
EndTime	object	End time of connection between EV and charger (HH:MM:SS)
StartDatetime	object	Start date time of connection between EV and charger (YYYY-MM-DD HH:MM:SS)

Column	Type	Description
EndDatetime	object	End date time of connection between EV and charger (YYYY-MM-DD HH:MM:SS)
Duration	int64	Charger connection duration (unit: minute)
Demand	float64	Amount of power charged to the EV (unit: kWh)

Cabe remarcar que los datos incluyen la hora de inicio y final de la sesión y la energía demandada en cada sesión. Sin embargo, para realizar la predicción de la demanda es necesario obtener la demanda agregada por hora o día, ya que hay que tener en cuenta la existencia de sesiones activas en una o varias franjas temporales.

3.2 Análisis Exploratorio de Datos (EDA)

El dataset utilizado no presentaba una alta complejidad al estar destinado a un estudio de series temporales univariable. Además, algunos de los atributos, como los temporales, eran redundantes. Sin embargo, siempre es necesario realizar un análisis de los datos previo a la realización de cualquier proyecto de ciencia de datos. Se utilizó la librería Pandas, de Python, para la manipulación y análisis de datos del EDA y también del feature engineering detallado en el siguiente capítulo.

En el jupyter notebook *preprocessed_data* se puede ver completamente el análisis realizado, pero a modo de resumen, solo se detallarán las conclusiones más destacadas.

- Los datos abarcan un espacio temporal de exactamente un año, del 30 de septiembre del 2021 al 30 de septiembre del 2022.

- Ninguno de los atributos tenía valores NaN.

- Las localizaciones con más muestras son public area (14,082) y apartment (14,038), mientras que las que tienen más cargadores son public area (437) y public institution (402).

- Analizando las demandas de las sesiones se detectó la existencia de sesiones con una duración negativa o 0. La incongruencia de estos datos podía afectar al cálculo de la demanda agregada por tiempo que se explicará en el siguiente capítulo, por lo que estas muestras se eliminaron del dataset.

4 Preprocessamiento de los datos (Feature Engineering)

Como se ha descrito anteriormente, para poder realizar las predicciones de la demanda de recarga horaria se necesita una serie temporal con la demanda agregada por tiempo. Para ello, se requieren una serie de transformaciones de preprocesamiento usando los datos disponibles. Estas transformaciones también se realizan en el jupyter notebook *preprocessed_data.*

La estrategia que se siguió en el proyecto es la estrategia Top-Down. Consiste en analizar los datos desde el mayor nivel de agregación y se crea una solución para luego ser comparada con niveles inferiores. Se ha considerado que los dos niveles de agregación razonables para la demanda es la horaria y la diaria, por lo que se calcularan soluciones para ambos niveles para luego elegir el nivel de agregación con el menor error.

4.1 Feature Engineering

El primer paso del feautre engineering ha sido pasar las columnas *StartDatetime* y *EndDatetime* de objeto a formato datetime de pandas para poder manipularas correctamente. Como la columna *Duration* estaba en minutos, se transformó a horas para el cálculo de la potencia media de la sesión. Finalmente, se calcula la potencia media de cada sesión dividiendo la columna *Demand*, en kWh, entre la columna *Duration*, en h. Esto se realiza usando el *Código 4.1*.

```
# String columns to datetime format
df['StartDatetime'] = pd.to_datetime(df['StartDatetime'])
df['EndDatetime'] = pd.to_datetime(df['EndDatetime'])

# Duration column from min to h
df['Duration'] = df['Duration']/60
df = df[['StartDatetime', 'EndDatetime', 'Demand', 'Duration']]

# Calculus of the power median per sesion
df['medPower'] = df['Demand'] / df['Duration']
```

***Código 4.1**: Cálculo de la potencia media de cada sesión*

Una vez obtenida la demanda media de cada sesión de recarga, para calcular la demanda agregada se realiza una discretización, horaria o diaria, del tiempo desde la primera sesión hasta la última. De esta forma se pueden dividir las sesiones en franjas y las que ocurran de manera simultánea, se sumarán. Además, se guardarán también el número de sesiones activas durante la franja de tiempo. En el *Código 4.2* se observa el código empelado para la

demanda agregada diaria, pero también se hizo con a nivel diario simplemente cambiando la frecuencia en el método *date_range*.

Finalmente, se guardan los datos en un nuevo dataframe y se exportan en archivos .csv que posteriormente serán utilizados para entrenar los modelo.

```
date_series_d = pd.date_range(start=df['StartDatetime'].min(),
                              end=df['EndDatetime'].max(), freq="1d")

val_sessions = []
val_total_power = []

for date in date_series_d:
    active_sessions = df[(df['StartDatetime'] <= date) & (df['EndDatetime']
                         >= date)]

    session_count = active_sessions.shape[0]
    total_power = active_sessions['medPower'].sum()

    val_sessions.append(session_count)
    val_total_power.append(total_power)

# Creating the new dataframe
final_d = pd.DataFrame({
    'date': date_series_d,
    'val_sessions': val_sessions,
    'val_total_power': val_total_power
})

# Export to csv
final_h.to_csv('./processed_data/data_h.csv', index=False)
final_d.to_csv('./processed_data/data_d.csv', index=False)
```

Código 4.2: *Cálculo de la demanda agregada y creación del nuevo dataframe*

4.2 Visualización de los datos

En el mismo Jupyter notebook (*preprocessed_data),* se han representado las dos series temporales de la demanda agregada mediante la librería matplotlib para observar su estacionalidad y variabilidad. La demanda agregada diaria se representa en la *Figura 4.1* y la horaria en la *Figura 4.2*.

En ambas graficas se puede observar una tendencia creciente de la serie, tanto en la demanda como en su variabilidad. En el caso de la demanda agregada horaria, el aumento de la variabilidad parece ser más visible que en el caso diario.

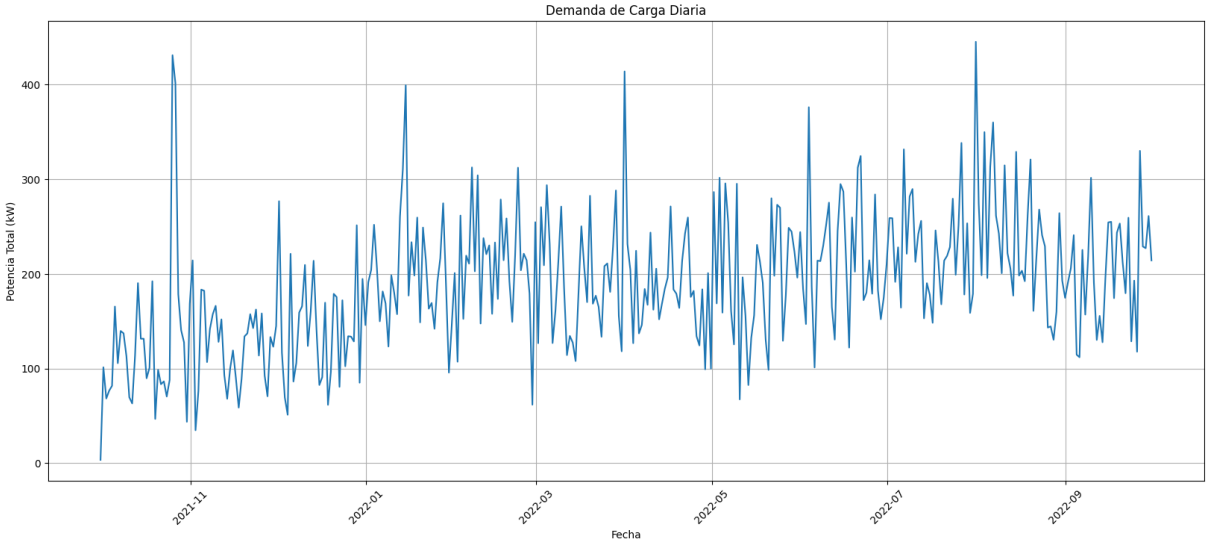

Figura 4.1: *Representación de la demanda agregada diaria*

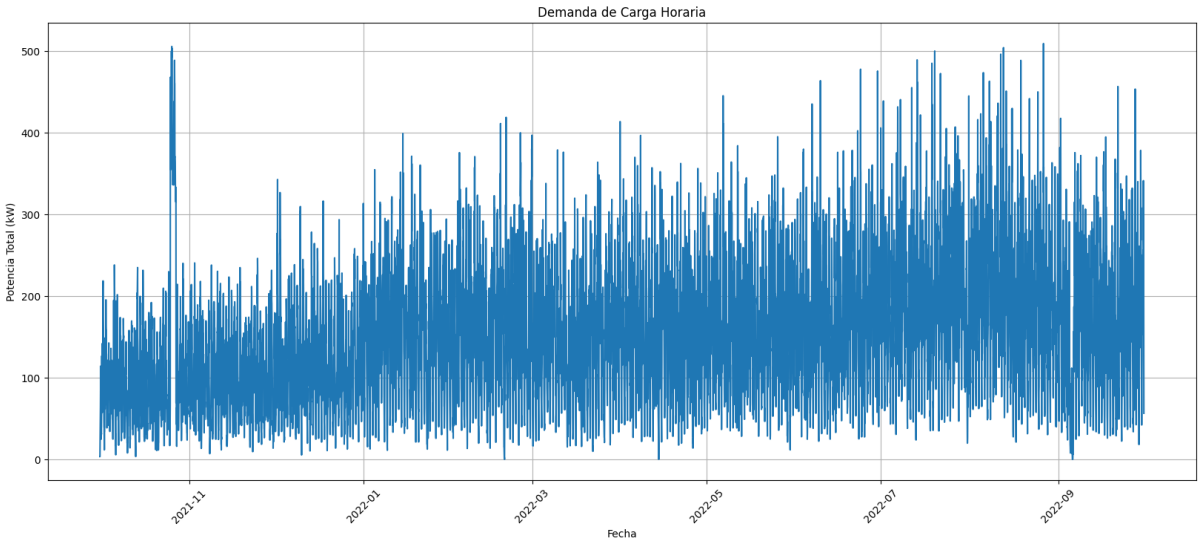

Figura 4.2*: Representación de la demanda agregada horaria*

Por otro lado, en ambas graficas se observa un pico de demanda puntual entorno al 26 de octubre de 2021. En la Figura 4.3 se puede ver representado en el caso de la demanda horaria de una forma más clara. Este pico se podría catalogar como un outlier o también como un evento significativo. Se intentó verificar si era debido a alguna festividad existente en Korea del Sur o alguna subida de tarifas o impuestos en la electricidad, sin embargo, no se pudo encontrar ninguna explicación. Finalmente, se decidió no tratar esta singularidad y mantenerla en la serie temporal.

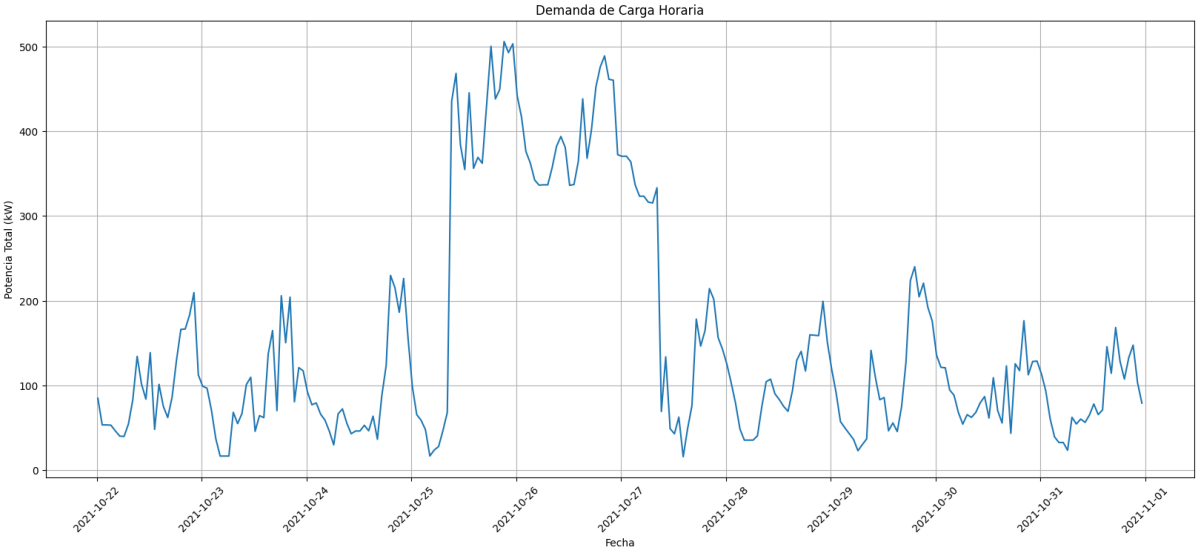

***Figura 4.3**: Representación del pico de demanda puntual*

5 Metodología

En este capítulo se explicará la metodología que se ha seguido a lo largo del proyecto, así como todos los procesos que se han realizado para entrenar los diferentes modelos, calcular las métricas y las predicciones.

5.1 Estrategia y flujo de trabajo

En este apartado se comentará, brevemente y sin entrar en detalles, la estrategia y los pasos seguidos en el proyecto para tener una rápida visión general del proyecto.

Como ya se ha comentado anteriormente, la estrategia seguida en este proyecto es la Top-down, por lo que primero se van a analizar las métricas de las soluciones obtenidas según su nivel de agregación (horario o diario) para luego escoger el mejor de los dos. Para realizar esta evaluación, se utilizaron dos tipos distintos de modelos con varias configuraciones de sus hiperparámetros. En el siguiente apartado se explicará más en detalle sobre los modelos y las configuraciones.

Luego, una vez decidido el nivel de agregación, se entrenarán 7 tipos de modelos, se evaluará su función de perdida y se representarán las predicciones sobre el conjunto de datos de entrenamiento, validación y test para escoger el modelo más eficiente. Este

Finalmente, se realizará un estudio más detallado sobre el error de ese modelo y también se realizarán predicciones a futuro que sean más adecuadas para el problema de la predicción de la demanda eléctrica ya que las realizadas durante la evaluación del modelo solo ofrecen un valor de salida, es decir, para el siguiente instante de tiempo.

En la Figura 5.1 se pueden ver todo el proceso representado en forma de diagrama.

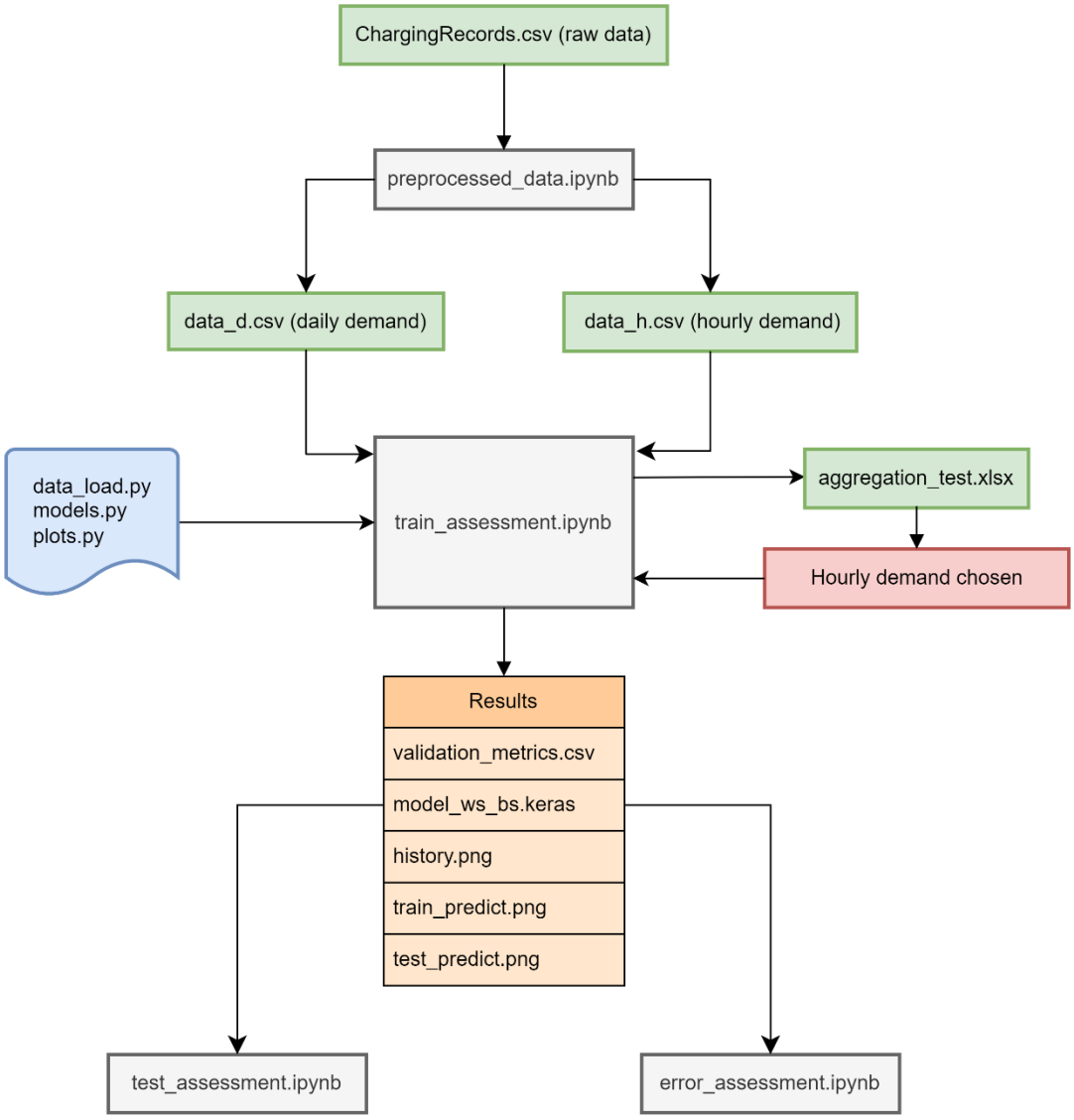

Figura 5.1: *Diagrama del flujo de trabajo*

5.2 Modelos

A priori, antes de empezar un proyecto, nunca se sabe qué modelo y qué arquitectura ofrecerá el mejor rendimiento, por lo que es fundamental probar distintas configuraciones para conseguir un modelo de alta precisión y adecuado para el problema que se le requiere solucionar.

A lo largo del proyecto se han utilizado varios modelos con diferentes arquitecturas y tipos de redes neuronales. En total, se han utilizado 7 tipos distintos de modelos, 5 de ellos contienen capas LSTM y 2 con capas LSTM Bidireccionales. Mas adelante se detallarán las características de cada uno de ellos con sus respectivas funciones y esquemas.

Para poder seleccionar el tipo de modelo de una forma rápida y sencilla se ha elaborado una función llamada get_model, la cual permite seleccionar el modelo de forma automática al introducir el nombre del modelo. Se muestra en el Código 5.1.

```
def get_model(name, window_size, num_features):
    if name == 'LSTM':
        return get_LSTM(window_size, num_features)
    elif name == 'LSTM_stacked':
        return get_LSTM_stacked(window_size, num_features)
    elif name == 'LSTM_stacked_2':
        return get_LSTM_stacked_2(window_size, num_features)
    elif name == 'LSTM_stacked_3':
        return get_LSTM_stacked_3(window_size, num_features)
    elif name == 'LSTM_stacked_3_alt':
        return get_LSTM_stacked_3_alt(window_size, num_features)
    elif name == 'Bidirectional_LSTM':
        return get_Bidirectional_LSTM(window_size, num_features)
    elif name == 'Bidirectional_LSTM_alt':
        return get_Bidirectional_LSTM_alt(window_size, num_features)
```

Código 5.1: *Función get_model para seleccionar el modelo*

5.2.1 Red neuronal recurrente LSTM (LSTM)

El primer modelo, LSTM, es el más sencillo de todos ya que esta solamente constituido por una capa LSTM de 64 celdas, además de la capa densa de 1 neurona que genera el valor de la predicción.

```
def get_LSTM(window_size, num_features):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(window_size, num_features)),
        tf.keras.layers.LSTM(64, return_sequences=False),
        tf.keras.layers.Dense(1),
        ])
    return model
```

Código 5.2: *Función para construir el modelo de red neuronal LSTM*

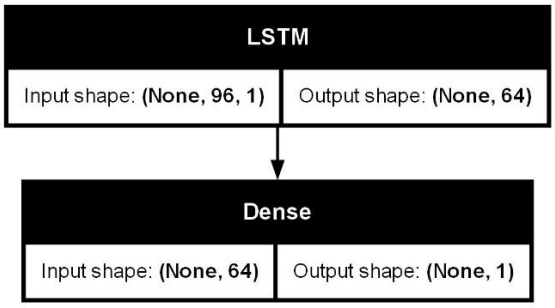

***Figura 5.2**: Esquema del modelo LSTM*

5.2.2 Red neuronal recurrente LSTM doble capa

Este modelo contiene dos capas LSTM de 32 celdas apiladas.

```
def get_LSTM_stacked(window_size, num_features):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(window_size, num_features)),
        tf.keras.layers.LSTM(32, return_sequences=True),
        tf.keras.layers.LSTM(32, return_sequences=False),
        tf.keras.layers.Dense(1),
        ])
    return model
```

***Código 5.3:** Función para construir el modelo de red neuronal LSTM doble capa*

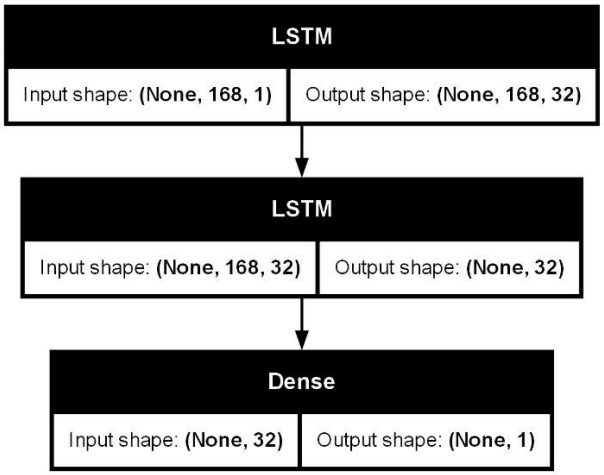

***Figura 5.3:** Esquema del modelo LSTM doble capa*

5.2.3 Red neuronal recurrente LSTM triple capa

Este modelo está constituido por 3 capas LSTM de 64 celdas cada una.

```
def get_LSTM_stacked_2(window_size, num_features):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(window_size, num_features)),
        tf.keras.layers.LSTM(64, return_sequences=True),
        tf.keras.layers.LSTM(64, return_sequences=True),
        tf.keras.layers.LSTM(64, return_sequences=False),
        tf.keras.layers.Dense(1),
        ])
    return model
```

Código 5.4: *Función para construir el modelo de red neuronal LSTM triple capa*

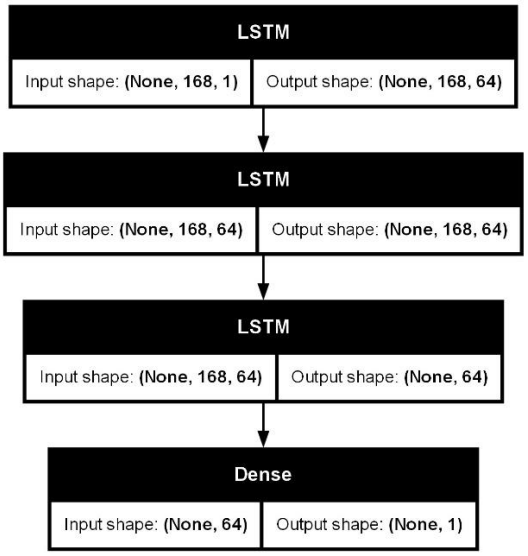

Figura 5.4: *Esquema del modelo LSTM triple capa*

5.2.4 Red neuronal recurrente LSTM multicapa

Este modelo ya tiene más complejidad ya que, aparte de tener 3 capas LSTM de 64 celdas cada una, también tiene 3 capas densas de 64 neuronas con funciones de activación Relu.

```
def get_LSTM_stacked_3(window_size, num_features):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(window_size, num_features)),
        tf.keras.layers.LSTM(64, return_sequences=True),
        tf.keras.layers.LSTM(64, return_sequences=True),
        tf.keras.layers.LSTM(64, return_sequences=False),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1)
    ])
    return model
```

Código 5.5: *Función para construir el modelo de red neuronal LSTM multicapa*

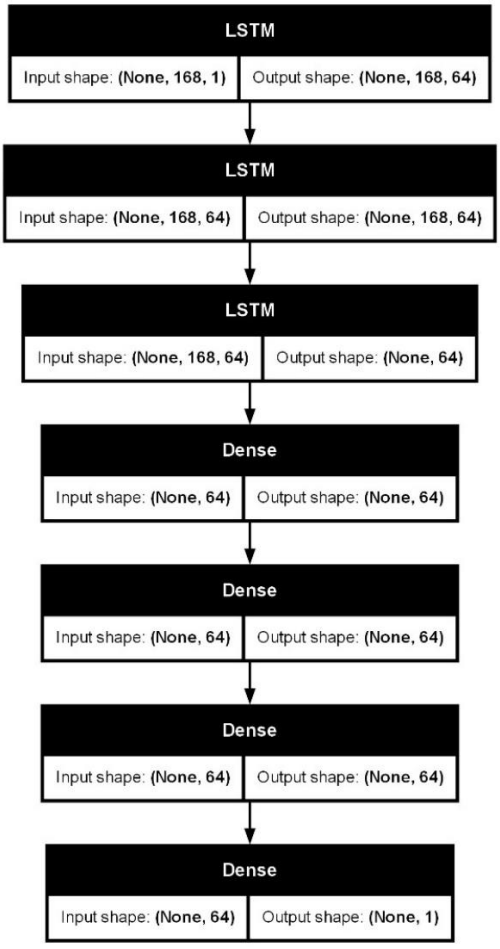

Figura 5.5: *Esquema del modelo LSTM multicapa*

5.2.5 Red neuronal recurrente LSTM multicapa alternada

Modelo parecido al anterior, pero en este caso las capas LSTM i densas se va alternando de forma que después de una capa LSTM hay una capa densa.

```
def get_LSTM_stacked_3_alt(window_size, num_features):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(window_size, num_features)),
        tf.keras.layers.LSTM(64, return_sequences=True),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.LSTM(64, return_sequences=True),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.LSTM(64, return_sequences=False),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1)
    ])
    return model
```

Código 5.6: *Función para construir el modelo de red neuronal LSTM multicapa alternada*

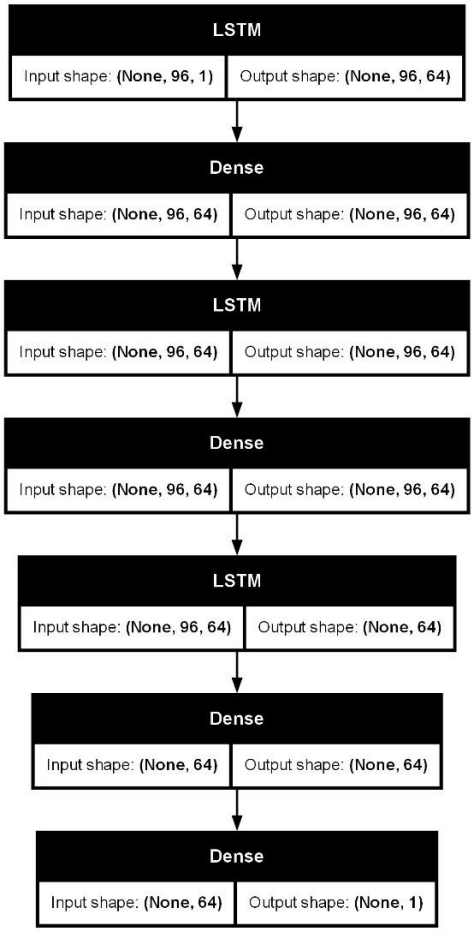

Figura 5.6: *Esquema del modelo LSTM multicapa alternada*

5.2.6 Red neuronal recurrente LSTM bidireccional multicapa

En este modelo se utilizan capas LSTM bidireccionales que podrán tener en cuenta valores del pasado y del futuro a la hora de calcular las activaciones de las neuronas. Las 3 capas LSTM Bidireccionales van seguidas de 3 capas densas.

```
def get_Bidirectional_LSTM(window_size, num_features):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(window_size, num_features)),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=False)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1)
    ])
    return model
```

Código 5.7: *Función para construir el modelo de red neuronal LSTM bidireccional multicapa*

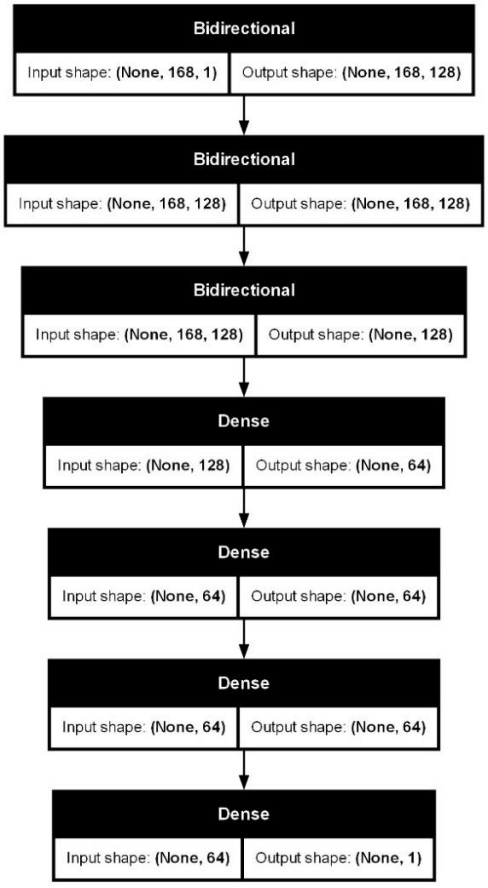

Figura 5.7: *Esquema del modelo LSTM bidireccional multicapa*

5.2.7 Red neuronal recurrente LSTM bidireccional multicapa alternada

Este modelo es similar al anterior que utiliza capas LSTM bidireccionales pero la capas LSTM bidireccionales y las capas densas se van alternando en vez de estar en dos bloques distintos.

```python
def get_Bidirectional_LSTM_alt(window_size, num_features):
    model = tf.keras.models.Sequential([
        tf.keras.layers.Input(shape=(window_size, num_features)),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=False)),
        tf.keras.layers.Dense(64, activation='relu'),
        tf.keras.layers.Dense(1)
    ])
    return model
```

Código 5.8: *Función para construir el modelo de red neuronal LSTM bidireccional multicapa alternado*

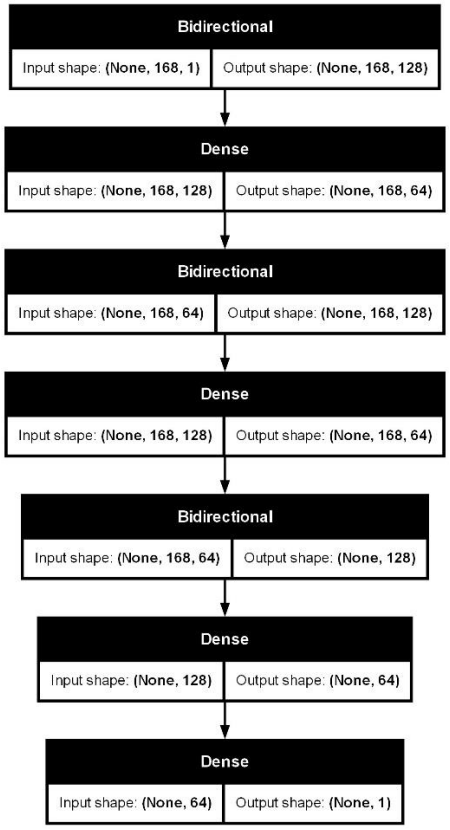

Figura 5.8: *Esquema del modelo LSTM bidireccional multicapa*

5.3 Entrenamiento de los modelos

Para el entrenamiento de los modelos se ha utilizado el jupyter notebook *train_assessment.ipynb* ya que de esta forma era más cómodo comparar y analizar los distintos modelos a lo largo del proyecto.

El primer paso que se realiza en el entrenamiento es la carga de los datos. Para ello, se creó una función llamada *load_dataset,* dentro del script *data_load.py,* que permite seleccionar el nivel de agregación de la data que queremos utilizar. Justo debajo se encuentra la función *get_h* que carga el dataset como dataframe y luego devuelve times y series. Times es una serie de pandas con las franjas horarias y series es la demanda eléctrica agregada almacenada en un array de numpy.

```
def load_dataset(name):
    if name == 'data_h':
        return get_h()
    elif name == 'data_h_public':
        return get_h_public()
    elif name == 'data_d':
        return get_d()
    else:
        return get_XXX()

def get_h():
    # Load the dataset
    file_path = "data_preprocessing/processed_data/data_h.csv"
    df = pd.read_csv(file_path)
    times = df['date']
    # Convert pandas dataframe to numpy array with float64
    series = df['val_total_power'].astype('float64').values

    return series,times
```

Código 5.9: *Función que carga los datos según el nivel de agregación*

5.3.1 Normalización de los datos

Una vez cargados los datos, estos deben ser normalizados antes de ser introducidos al modelo, ya que permite que el descenso del gradiente sea más estable y eficiente por lo que mejora la convergencia del entrenamiento.

En este caso se normalizan los datos de forma lineal entre 0.1 y 1. No se ha normalizado entre 0 y 1 ya que los valores cercanos a 0 pueden llegar a saturar las funciones de activación

sigmoide y tanh, que son las funciones de activación presentes en una celda LSTM. La normalización se ha implementado mediante la función MinMaxScaler de la librería sklearn, como se muestra en el Código 5.

La fórmula es la siguiente:

$$y = \frac{x - x_{min}}{x_{max} - x} \tag{6.1}$$

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler(feature_range=(0.1,1))
series_scaled = scaler.fit_transform(series)
```

Código 5.10: *Normalización de los datos*

5.3.2 División de los datos en entrenamiento, validación y test

Como en la mayoría de los proyectos de machine learning, el conjunto de datos se divide en datos de entrenamiento, validación y test. Los datos de entrenamiento se utilizan para ajustar los pesos y el sesgo durante el proceso de aprendizaje. Los datos de validación se utilizan para afinar el modelo, ayudando a seleccionar hiperparámetros óptimos como el batch size o el window size (en el caso de series temporales) y también evitar el overfitting. El conjunto de test se utiliza para evaluar el rendimiento del modelo una vez entrenado y con los hiperpárametros ajustados.

```
train_percent = 0.7
val_percent = 0.2
test_percent = 0.1

train_size = int(len(series) * train_percent)
val_size = int(len(series) * val_percent)

x_train = series[:train_size]
x_train_scaled = series_scaled[:train_size]

x_val = series[train_size:train_size + val_size]
x_val_scaled = series_scaled[train_size:train_size + val_size]

x_test = series[train_size + val_size:]
x_test_scaled = series_scaled[train_size + val_size:]
```

Código 5.11 *Separación del conjunto de datos en train, validation y test.*

La importancia de tener un conjunto de test que no haya sido utilizado por el modelo previamente radica en que los datos de validación, aunque no hayan sido utilizados en el entrenamiento del modelo, añaden cierto sesgo al ser utilizados para ajustar los hiperparámetros.

Se decide separar los datos de la siguiente manera: 70% entrenamiento, 20% validación y 10% test. En el Código 5.11 se realiza la separación del conjunto de datos.

5.3.3 Creación de los generadores secuenciales

Las series temporales requieren un tratamiento especial antes de entrenar al modelo debido a la dependencia temporal de los datos. Por lo tanto, es necesario transformar la serie temporal unidimensional en secuencias de datos de entrada.

Esto se puede realizar de manera sencilla mediante el uso de generadores de series secuenciales. En este caso, se ha utilizado la función *TimeseriesGenerator* de la librería *Keras.preproecessing.sequence.* Para modelar los generadores es necesario especificar los datos de entrada y salida, la longitud de las secuencias de entrada (window size) y el número de muestras que se procesan antes de que el modelo actualice sus pesos en el entrenamiento (batch size).

La creación de los generadores de entrenamiento, validación y test se realiza en el Código 5.12.

```
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator

batch_size = 32
window_size = 24 * 4 # 4 days

train_generator = TimeseriesGenerator(x_train_scaled,
                                      x_train_scaled,
                                      length=window_size,
                                      batch_size=batch_size)

validation_generator = TimeseriesGenerator(x_val_scaled,
                                           x_val_scaled,
                                           length=window_size,
                                           batch_size=batch_size)

test_generator = TimeseriesGenerator(x_test_scaled,
                                           x_test_scaled,
                                           length=window_size,
                                           batch_size=batch_size)
```

Código 5.12: *Creación de los generadores secuenciales*

5.3.4 Carga y compilación del modelo

El modelo se carga mediante la función *get_model* que se ha explicado en el inicio del capítulo 5.2. Los argumentos de la función son el nombre del modelo, el tamaño de la secuencia de entrada y la dimensión de los valores de la secuencia, en este caso será igual a 1 ya que la serie temporal es univariable.

```
num_features = 1
model_name = 'LSTM_stacked_3_alt'
model = get_model(model_name,
                  window_size,
                  num_features)
```

***Código 5.13**: Proceso de carga del modelo*

El siguiente paso es la compilación del modelo. En la compilación se especifica la función de coste utilizada, el optimizador y las métricas de evaluación que se calcularan durante el entrenamiento. En este caso se utilizó la función de perdida Huber ya que es una combinación de las funciones MAE (Mean Absolute Error) y MSE (Mean Squared Error), siendo cuadrática para errores pequeños y lineal para errores grandes, minimizando así las penalizaciones sufridas en las otras dos funciones de perdida. Respecto al optimizador, se utilizó el optimizador Adam mientras que las métricas utilizadas son precisamente el MAE y el MSE.

```
optimizer = 'adam'
metrics = ['mae','mse']

model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=metrics)
```

***Código 5.14:** Compilación del modelo*

5.3.5 Regularizaciones (Callbacks)

Además de las configuraciones ya explicadas para el entrenamiento, se utilizaron otras herramientas para controlar el proceso de entrenamiento. La primera de ellas es el EarlyStopping, que consiste en detener el entrenamiento si la métrica referenciada no mejora después de un número específico de epochs, evitando el sobreajuste del modelo. En este caso, la métrica referenciada es el valor de la función de perdida de los datos de validación y el número de épocas sin mejora para detener el entrenamiento fue de 5.

Por otro lado, también se utilizó la función ModelCheckpoint para guardar el mejor modelo según la métrica referenciada. La métrica referenciada fue la misma que la que se utilizó en el Early Stopping.

Ambas funciones se integraron dentro de una función llamada *configure_callbacks* que se encuentra dentro del script *models.py* y se puede observar en el Código 5.15.

```
def configure_callbacks(early_stopping = False,patience=10, checkpoint=
                  False, model_name= None, window_size=None, batch_size=None):
    callbacks = []
    if checkpoint:
        model_nm = model_name + f'_ws{window_size}_bs{batch_size}'
        model_name += f'_ws{window_size}_bs{batch_size}.keras'
        checkpoint_dir = f'Results/train_{model_nm}_results'
        checkpoint_filepath = os.path.join(checkpoint_dir, model_name)
        os.makedirs(checkpoint_dir, exist_ok=True)
        model_checkpoint_callback = ModelCheckpoint(
            filepath=checkpoint_filepath,
            save_weights_only=False,
            monitor='val_loss',
            mode='min',
            save_best_only=True)
        callbacks.append(model_checkpoint_callback)

    if early_stopping:
        callbacks.append(EarlyStopping(patience=patience, monitor='val_loss'))

    return callbacks

callbacks = configure_callbacks(early_stopping = True,
                                patience=10,
                                checkpoint= True,
                                model_name= model_name,
                                window_size= window_size,
                                batch_size= batch_size
                                )
```

Código 5.15: *Configuración de los callbacks*

5.3.6 Entrenamiento del modelo

Una vez definidos todos los pasos anteriores, ya se puede proceder con el entrenamiento del modelo. Para ello se van a utilizar el método fit de Keras, donde se van a utilizar como argumentos los generadores de entrenamiento y validación, el número de épocas deseado y

los callbacks ya definidos. El número de épocas es el número de pasos completos que se realizan sobre los datos de entrenamiento. En este caso se fijó a 30.

```
epochs = 30
history = model.fit(train_generator,
                    verbose=1,
                    epochs=epochs,
                    validation_data=validation_generator,
                    callbacks=callbacks)
```

Código 5.16: *Entrenamiento del modelo*

5.4 Validación y afinamiento del modelo

Una vez que el modelo ha sido entrenado, se pueden hacer predicciones utilizando datos de validación, es decir, datos que no se usaron durante el entrenamiento. Esto permite evaluar si el modelo puede aplicar lo que ha aprendido a datos nuevos que no ha visto antes. Este proceso ayuda a confirmar que el modelo está generalizando el conocimiento adquirido durante el entrenamiento, en lugar de simplemente memorizar y replicar los datos.

Para evaluar el rendimiento del modelo, se puede hacer una comparación visual entre las predicciones del modelo y los valores reales. Sin embargo, para obtener una evaluación cuantitativa precisa, se utilizan diversas métricas que miden el error entre las predicciones y los datos reales. En este trabajo, se emplean la raíz del error cuadrático medio (RMSE) y el error absoluto medio (MAE) como métricas de evaluación. Estos errores se calcularán tanto para las predicciones en los datos de entrenamiento como para las de validación, comparándolos con sus valores reales correspondientes. Además, estas se registrarán todas en el csv llamado *validation_metrics.csv*.

Este proceso se lleva a cabo en el Código 5.17.

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j| \tag{6.2}$$

$$RMSE = \sqrt{\sum_{j=1}^{n}\frac{(y_j - \hat{y}_j)^2}{n}} \tag{6.3}$$

En resumen, la validación es el proceso mediante el cual se evalúa el rendimiento del modelo. Esta etapa es crucial, ya que los resultados de la validación y el análisis del entrenamiento se utilizan para determinar qué modificaciones son necesarias para mejorar el modelo. Los modelos suelen tener numerosos hiperparámetros, como la arquitectura del modelo, el número de épocas, el optimizador, y el tamaño de lote (batch size). Por ello, una estrategia adecuada para ajustar estos hiperparámetros puede ahorrar mucho tiempo en el entrenamiento de los modelos.

El flujo de trabajo en proyectos de este tipo a menudo implica un ciclo de entrenamiento, validación, ajuste de hiperparámetros y repetición de estas iteraciones hasta lograr un modelo con un rendimiento óptimo.

```
train_predict = predict(model,
                        train_generator,
                        scaler)

val_predict = predict(model,
                      validation_generator,
                      scaler)

train_rmse = root_mean_squared_error(x_train[window_size:,0], train_predict[:,0])
val_rmse = root_mean_squared_error(x_val[window_size:,0], val_predict[:,0])
train_mae = mean_absolute_error(x_train[window_size:,0], train_predict[:,0])
val_mae = mean_absolute_error(x_val[window_size:,0], val_predict[:,0])

results = {
    'Model': [model_name] * 4,
    'Window Size': [window_size] * 4,
    'Batch Size': [batch_size] * 4,
    'Metric': ['RMSE', 'RMSE', 'MAE', 'MAE'],
    'Data Split': ['Train', 'Val', 'Train', 'Val'],
    'Score': [train_rmse, val_rmse, train_mae, val_mae]
}
results_df = pd.DataFrame(results)
csv_file_path = 'Results/validation_metrics.csv'

# Checking if the file already exists
if os.path.exists(csv_file_path):
    results_df.to_csv(csv_file_path, mode='a', header=False, index=False)
else:
    results_df.to_csv(csv_file_path, index=False)
```

Código 5.17: *Fase de validación y recopilación de las métricas*

5.5 Test

La fase de test es similar a la fase de validación, pero sin la intención de ajustar o volver a entrenar el modelo. Su objetivo es hacer predicciones utilizando un conjunto de datos que es independiente de los datos de entrenamiento para evaluar el rendimiento final del modelo.

Durante esta fase, se realizan predicciones sobre los datos de entrenamiento, sobre los datos de test y también se efectúan predicciones para varios pasos hacia el futuro.

5.5.1 Predicción sobre el conjunto de entrenamiento y de test

El modelo utiliza una secuencia de entrada con el tamaño de la ventana temporal definida y genera como salida el valor del siguiente instante de tiempo correspondiente a esa secuencia de entrada. Estas predicciones se realizan tanto en el conjunto de entrenamiento como en el conjunto de test. A continuación, se crean gráficos que muestran todas las predicciones, junto con gráficos ampliados que permiten observar las predicciones con mayor detalle. El Código 5.18 es el utilizado.

Las funciones *plot_predictions* y *raw_data_vs_predictions* junto con todas las demás funciones que se han utilizado para representar gráficos, se encuentran en el script *plots.py*.

```
# Plotting the whole dataset
plot_predictions(train_predict, val_predict, test_predict, window_size,
                 times, series, x_train, x_val)

# Plotting training data and train predictions
plt.figure(figsize=(20, 8))
plt.plot(x_train[window_size:])
plt.plot(train_predict[:,0])

# Plotting test data and test predictions
plt.figure(figsize=(20, 8))
plt.plot(x_test[window_size:])
plt.plot(test_predict[:,0])

# Train zoom plot
start = 5350
end = 5650
raw_data_vs_predictions(x_train, train_predict, window_size, start, end, 'Train')
plt.savefig(f'Results/train_{model_name}_ws{window_size}_bs{batch_size}_results/t
rain_predict.png')

# Test zoom plot
start = 200
end = 500
raw_data_vs_predictions(x_test, test_predict, window_size, start, end, 'Test')
plt.savefig(f'Results/train_{model_name}_ws{window_size}_bs{batch_size}_results
/test_predict.png')
```

Código 5.18: *Representación de las predicciones*

5.5.2 Predicción a futuro

Los modelos descritos anteriormente están diseñados para predecir el siguiente valor en el tiempo a partir de una secuencia de entrada. Sin embargo, para abordar el problema de optimización de la gestión de la estación, se requiere prever varios valores futuros, no solo un único valor de salida, sino una secuencia de valores. Hay varias formas de lograr esto.

En este caso, se implementa un enfoque que utiliza un bucle para generar nuevas predicciones a partir de las predicciones anteriores, permitiendo así obtener múltiples valores de salida a partir de la misma secuencia de entrada.

Para este propósito, se define una ventana temporal que inicialmente contiene los valores de la secuencia de entrada. Con estos valores, se realiza la primera predicción. Luego, la ventana temporal se desplaza un paso hacia adelante, eliminando el primer valor de la secuencia original e incorporando la predicción generada en el paso anterior. Esta nueva secuencia de entrada se utiliza para generar otra predicción. La cantidad de veces que se repiten estos pasos determinará el número de valores futuros que se predecirán.

En el *Código 5.19* se muestra la función *forecast_and_evaluate* que permite visualizar las predicciones a futuro y también calcula los errores relativos entre las predicciones y los valores reales.

```
def forecast_and_evaluate(start, x_test, x_test_scaled, model, scaler,
                          window_size, num_features, horizon, plot=True):
    prediction = []

    current_batch = x_test_scaled[start : start + window_size]
    current_batch = current_batch.reshape(1, window_size, num_features)

    for i in range(horizon):
        current_pred = model.predict(current_batch)[0]
        prediction.append(current_pred)
        current_batch = np.append(current_batch[:, 1:, :], [[current_pred]],
                                  axis=1)

    rescaled_prediction = scaler.inverse_transform(prediction)
    real_data = x_test[start + window_size : start + window_size + horizon]

    sum_predictions = np.sum(rescaled_prediction)
    sum_real_data = np.sum(real_data)

    relative_error = (np.abs(sum_predictions - sum_real_data) / sum_real_data)
                      * 100
```

```
    if plot:
        plt.figure(figsize=(20, 8))
        plt.title('24h Horizon Forecast')
        plt.plot(real_data, label='Actual Data')
        plt.plot(rescaled_prediction, label='Predictions')
        plt.legend()
        plt.show()

    return rescaled_prediction, real_data, relative_error
```

Código 5.19: *Función para predecir más de un valor en el futuro y calcular el error relativo*

Este método de predicción a futuro se realiza en el jupyter notebook *test_assessment.ipynb*.

La función anterior viene definida por el parámetro *horizon* que marca el número de pasos temporales que realizara la función. Como se van a analizar las predicciones diarias, el valor de *horizon* será 24 ya que los datos son horarios.

Además de realizar predicciones diarias, también se ha utilizado la función anterior para establecer un bucle que permita generar predicciones desde un punto inicial a un punto final del conjunto de test. En este caso, tras cada iteración de 24 pasos el batch se reseteará con los datos de test para tener una mayor consistencia en la evaluación, evitar la acumulación de errores de las predicciones anteriores y controlar la variabilidad que puede surgir en un intervalo muy amplio.

En el *Código 5.20* se puede ver un ejemplo en que se hacen predicciones de 7 días consecutivos.

```
max_start = 144 # Last start value (7 days in this case)

relative_errors = []
all_predictions = []
all_real_data = []

for start in range(0, max_start + 1, 24):
    rescaled_prediction, real_data, relative_error =
    forecast_and_evaluate(start, x_test, x_test_scaled, model, scaler,
    window_size, num_features, horizon, plot=False)

    relative_errors.append(relative_error)
    all_predictions.extend(rescaled_prediction)
    all_real_data.extend(real_data)

all_predictions = np.array(all_predictions)
all_real_data = np.array(all_real_data)
```

```
plt.figure(figsize=(20, 8))
plt.title(f'Forecast for {(max_start // 24) +1} days')
plt.plot(all_real_data, label='Actual Data')
plt.plot(all_predictions, label='Predictions')
plt.legend()
plt.show()

for i, start in enumerate(range(0, max_start + 1, 24)):
    print(f'Dia {(start // 24) +1}, Error relativo: {relative_errors[i]:.2f}%')
```

Código 5.20: *Implementación de la función en bucle para 7 días consecutivos*

Finalmente, en el notebook *test_assessement.ipynb*, también se ha añadido un código basado en un bucle que compara las predicciones de los distintos modelos evaluados en el proyecto, de manera que al final quede una gráfica con las distintas predicciones superpuestas. En el *Código 5.21* se puede ver la implementación del bucle para un *horizon* de 24 horas.

```
start = 0
model_names = ['LSTM','LSTM_stacked','LSTM_stacked_2','LSTM_stacked_3',
               'LSTM_stacked_3_alt','Bidirectional_LSTM',
               'Bidirectional_LSTM_alt']

plt.figure(figsize=(20, 8))

for model_name in model_names:
    model_path =
f'Results/train_{model_name}_ws{window_size}_bs{batch_size}_results/{model_name
}_ws{window_size}_bs{batch_size}.keras'
    model_comp = tf.keras.models.load_model(model_path, compile=False)

    current_batch = x_test_scaled[start : start + window_size]
    current_batch = current_batch.reshape(1, window_size, num_features)

    prediction=[]
    for i in range(horizon):
        current_pred = model_comp.predict(current_batch)[0]
        prediction.append(current_pred)
        current_batch =
np.append(current_batch[:,1:,:],[[current_pred]],axis=1)

    rescaled_predictions = scaler.inverse_transform(prediction)

    plt.plot(rescaled_predictions, label=model_name)
```

```
plt.plot(x_test[start + window_size: start + window_size + horizon], '--',
label='Actual Data')
plt.legend()
plt.title('Model Comparison')
plt.show()
```

Código 5.21: *Bucle para comparar los distintos modelos*

6 Resultados

6.1 Selección del nivel de agregación

La primera decisión que se debía tomar en el proyecto era la selección del nivel de agregación. Como se explicó anteriormente, se calcularán soluciones para ambos niveles de agregación y el que presente los errores más bajos será el elegido. Para el estudio del nivel de agregación se utilizaron los dos primeros modelos (LSTM y LSTM_stacked) ya que son los más simples. Además, se evaluaron diferentes valores de los hiperparámetros. En el caso de la window size, se utilizaron ventanas temporales de 24, 96 y 168 horas, es decir 1, 4 y 7 días respectivamente, mientras que los tamaños de batch size analizados fueron 16, 32 y 64.

Tras realizar todos los entrenamientos de todas las combinaciones posibles, se anotaron los valores de la época con menor función de perdida en la validación. En este caso solo se utilizó la métrica MAE.

En la Tabla 6.1 se pueden observar los resultados. La tabla se puede encontrar en el Excel *aggregation_test.xlsx*.

Tabla 6.1: *Comparación de los niveles de agregación diarios y horarios*

Datos	Modelo	Window size	Batch	Train Loss	Val Loss	Train MAE	Val MAE
Diarios	LSTM	1 day	16	0.0091	0.0105	0.1073	0.1076
Diarios	LSTM	1 day	32	0.009	0.0105	0.1069	0.108
Diarios	LSTM	1 day	64	0.01	0.01	0.1135	0.1055
Diarios	LSTM	4 days	16	0.0081	0.0093	0.0989	0.1057
Diarios	LSTM	4 days	32	0.0089	0.0099	0.1039	0.1076
Diarios	LSTM	4 days	64	0.0088	0.0106	0.1044	0.1102
Diarios	LSTM	7 days	16	0.0084	0.0091	0.1031	0.1061
Diarios	LSTM	7 days	32	0.0087	0.0094	0.1	0.1075
Diarios	LSTM	7 days	64	0.0082	0.0092	0.0988	0.1069
Diarios	LSTM_stacked	1 day	16	0.0085	0.0102	0.103	0.1063
Diarios	LSTM_stacked	1 day	32	0.0088	0.0106	0.1052	0.1088
Diarios	LSTM_stacked	1 day	64	0.0101	0.0102	0.115	0.1066
Diarios	LSTM_stacked	4 days	16	0.0116	0.0094	0.1125	0.106
Diarios	LSTM_stacked	4 days	32	0.0111	0.01	0.1128	0.1077
Diarios	LSTM_stacked	4 days	64	0.0085	0.0096	0.1008	0.1067
Diarios	LSTM_stacked	7 days	16	0.0092	0.0105	0.1033	0.1116
Diarios	LSTM_stacked	7 days	32	0.0082	0.0988	0.1086	0.1081
Diarios	LSTM_stacked	7 days	64	0.0104	0.009	0.1105	0.1056
Horarios	LSTM	1 day	16	0.0033	0.005	0.0586	0.0744
Horarios	LSTM	1 day	32	0.0033	0.0054	0.0579	0.0809
Horarios	LSTM	1 day	64	0.0032	0.0054	0.0583	0.0749

Horarios	LSTM	4 days	16	0.0029	0.0049	0.0545	0.0719
Horarios	LSTM	4 days	32	0.003	0.0052	0.0563	0.0734
Horarios	LSTM	4 days	64	0.0031	0.0051	0.0567	0.0733
Horarios	LSTM	7 days	16	0.0032	0.0049	0.0566	0.0727
Horarios	LSTM	7 days	32	0.0029	0.0051	0.0542	0.0727
Horarios	LSTM	7 days	64	0.003	0.0054	0.0569	0.0757
Horarios	LSTM_stacked	1 day	16	0.003	0.0053	0.055	0.0764
Horarios	LSTM_stacked	1 day	32	0.0031	0.0049	0.056	0.0735
Horarios	LSTM_stacked	1 day	64	0.0033	0.0051	0.0595	0.0762
Horarios	LSTM_stacked	4 days	16	0.0031	0.005	0.0557	0.0738
Horarios	LSTM_stacked	4 days	32	0.003	0.0051	0.0571	0.0739
Horarios	LSTM_stacked	4 days	64	0.0032	0.0049	0.057	0.0725
Horarios	LSTM_stacked	7 days	16	0.0031	0.0051	0.0595	0.0762
Horarios	LSTM_stacked	7 days	32	0.0031	0.0054	0.0571	0.0758
Horarios	LSTM_stacked	7 days	64	0.0033	0.0049	0.0572	0.0726

Observando la función de perdida y el MAE es evidente que los datos horarios tienen un mejor rendimiento en todas las combinaciones posibles. Mientras que con los datos horarios la función de perdida oscila entre 0.0029 – 0.0033 y el MAE entre 0.07 – 0.08, con los datos diarios los valores oscilan entre 0.009 – 0.01 y 0.1 – 0.11 respectivamente. Por lo tanto, se seleccionaron los datos con el nivel de agregación horaria.

6.2 Entrenamiento

En la parte de entrenamiento se utilizaron todos los tipos de modelos descritos anteriormente con las distintas combinaciones que se pueden obtener al utilizar ventanas temporales de 24, 96 y 168 horas y tamaños de batch size de 16, 32 y 64 muestras. Además, todos los entrenamientos han sido realizados con 30 épocas.

Para cada combinación única de los modelos, se han almacenado en su carpeta correspondiente los siguientes archivos:

- El modelo entrenado en formato .keras
- El historial del entrenamiento con la evolución de las métricas y la función de perdida a lo largo de las épocas
- El gráfico de un fragmento de las predicciones del conjunto de entrenamiento
- El gráfico de un fragmento de las predicciones del conjunto de test

Como se explicó anteriormente, las métricas MAE y RSME de todos los entrenamientos se guardaron en un csv llamado *validation_metrics* dentro de la carpeta Results. Dentro de esta

carpeta también se han guardado las representaciones esquemáticas de los 7 modelos obtenidas utilizando la función *plot_model* de la librería *tensorflow.keras.utils.*

En total se han entrenado 63 modelos distintos por lo que presentar los resultados de todos y cada uno de ellos no sería muy eficiente. Por ello, solo se van a presentar los historiales de entrenamiento de los 7 modelos con un batch size de 32 y una window size de 96. Se pueden observar en las Figuras 6.1 hasta la 6.7.

En todos ellos se observa como la función de perdida y las metricas MAE y RSME disminuyen con el paso de las épocas. Como era esperable, en todas la graficas el error de entrenamiento es menor que el de validación. Por otro lado, se puede observar como el error de validación ha ido disminuyendo conforme lo ha hecho el error de entrenamiento y se ha estabilizado sin volver a incrementar el error. Esto es una señal de que el modelo no tiene overfitting ya que los callbacks están evitando prolongar el entrenamiento en el caso que sea necesario.

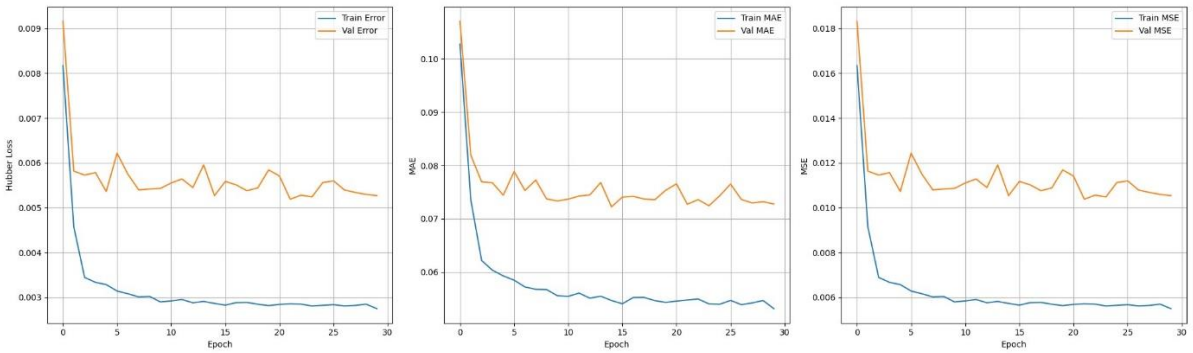

Figura 6.1: *Historial de entrenamiento del modelo LSTM (ws = 96, bs = 32)*

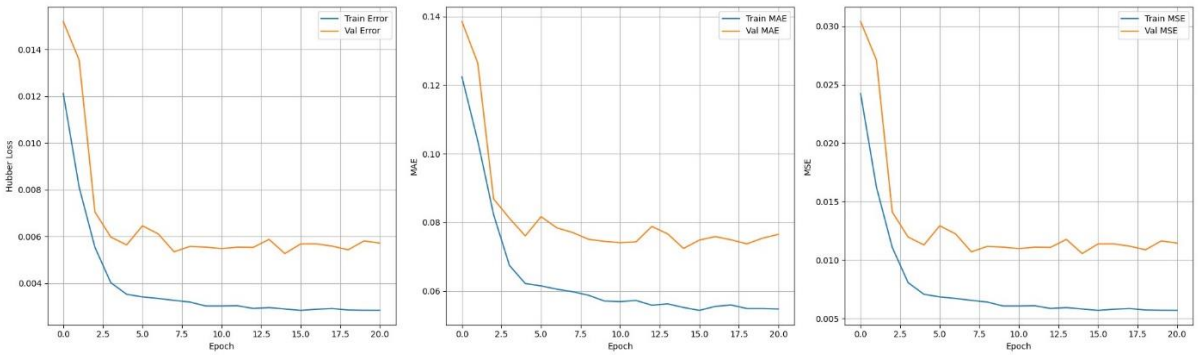

Figura 6.2: *Historial de entrenamiento del modelo LSTM_stacked (ws = 96, bs = 32)*

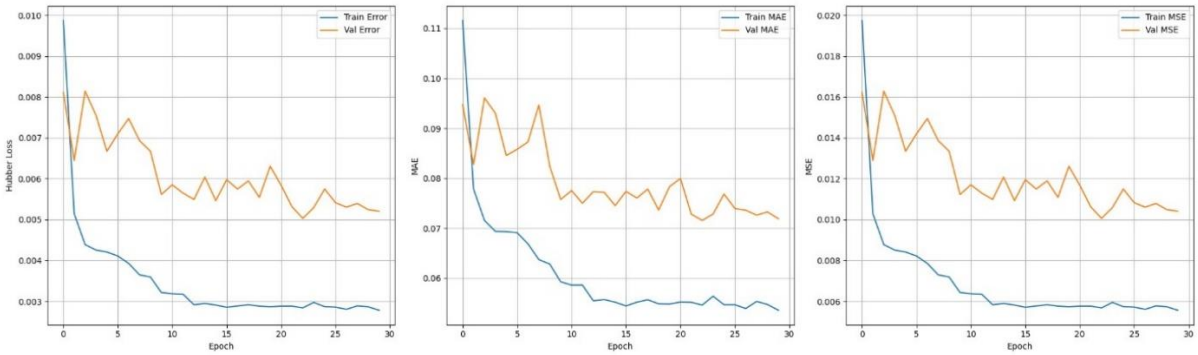

Figura 6.3: *Historial de entrenamiento del modelo LSTM_stacked_2 (ws = 96, bs = 32)*

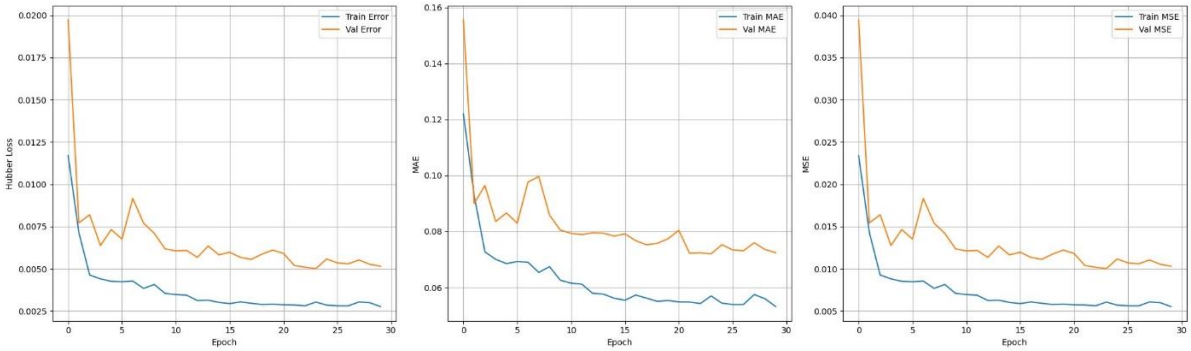

Figura 6.4: *Historial de entrenamiento del modelo LSTM_stacked_3 (ws = 96, bs = 32)*

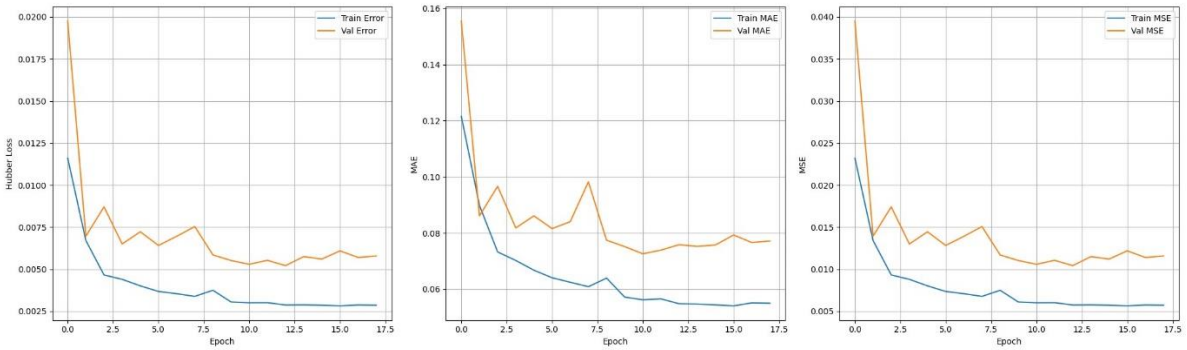

Figura 6.5: *Historial de entrenamiento del modelo LSTM_stacked_3_alt (ws = 96, bs = 32)*

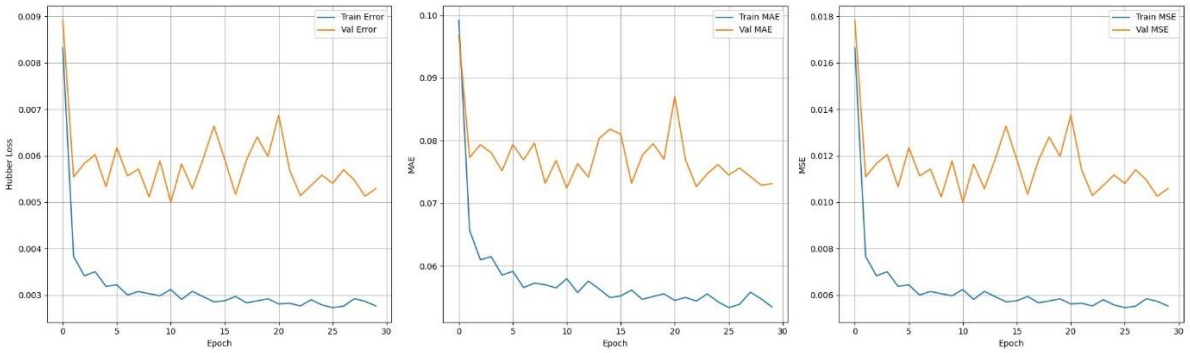

Figura 6.6: *Historial de entrenamiento del modelo Bidirectional_LSTM (ws = 96, bs = 32)*

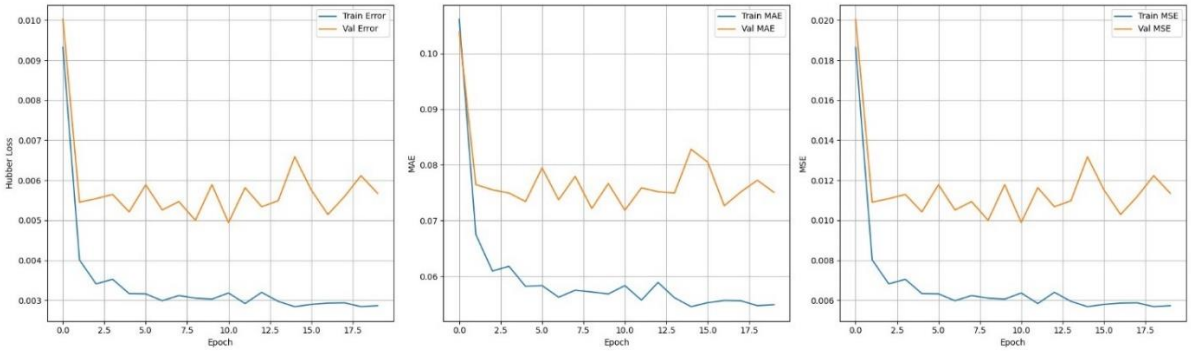

Figura 6.7: *Historial de entrenamiento del modelo Bidirectional_LSTM_alt (ws = 96, bs = 32)*

6.3 Validación

Analizando el *validation_metrics*, se observó que en el caso de la métrica MAE, su valor en las predicciones de entrenamiento oscila entre 28 y 33, mientras que el RMSE oscila entre 40 y 45. Por otro lado, en las predicciones de validación, el MAE oscila entre 38 y 43 mientras que el RMSE lo hace entre 52 y 57.

Debido a la gran cantidad de modelos y la relativamente poca variabilidad que existe en las métricas, se ha decidido no añadir ninguna tabla en la memoria. Comentando a nivel general, no hubo mucha diferencia entre los modelos y sus diferentes hiperparámetros, sin embargo, sí se pudo ver una ligera reducción de los errores en los modelos más complejos. Respecto a la window size, se observó que el rendimiento variaba entre modelos y no había un tamaño óptimo. En cambio, con el batch size si se vio una ligera mejora general del error con los tamaños de 16 y 32 muestras.

Los 3 modelos que mejor rendimiento tuvieron en validación fueron:

- *Bidirectional_LSTM_alt (ws = 96 y bs = 16)*
- *LSTM_stacked_3_alt (ws = 96 y bs = 16)*
- *LSTM_stacked_3 (ws = 96 y bs = 32)*

Considerando que los valores de la serie temporal original oscilan entre 0 y 500 kW, los valores de MAE y RMSE indican errores relativamente significativos. Sin embargo, la magnitud de estos errores debe ser evaluada en función del contexto y el enfoque adoptado para la implementación del modelo.

6.4 Test

Para realizar todos los análisis de prueba, se ha seleccionado uno de los tres mejores modelos, específicamente el modelo **LSTM_stacked_3** con un tamaño de ventana (*ws*) de 96 y un tamaño de lote (*bs*) de 32.

En primer lugar, las Figuras 6.8 y 6.9 muestran un fragmento de las predicciones tanto del conjunto de entrenamiento como del conjunto de prueba. Parece ser que el modelo se adapta relativamente bien a los datos de entrenamiento y test, pero aún le falta para capturar todos los patrones existentes. Además, parece que le cuesta predecir los valores pico de cada día.

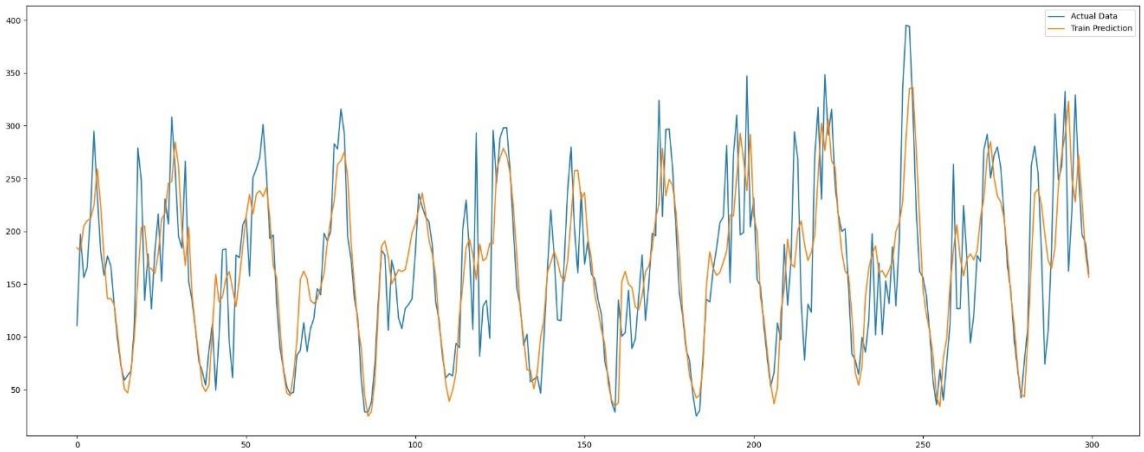

Figura 6.8: *Fragmento de las predicciones del conjunto de entrenamiento del modelo LSTM_stacked_3*

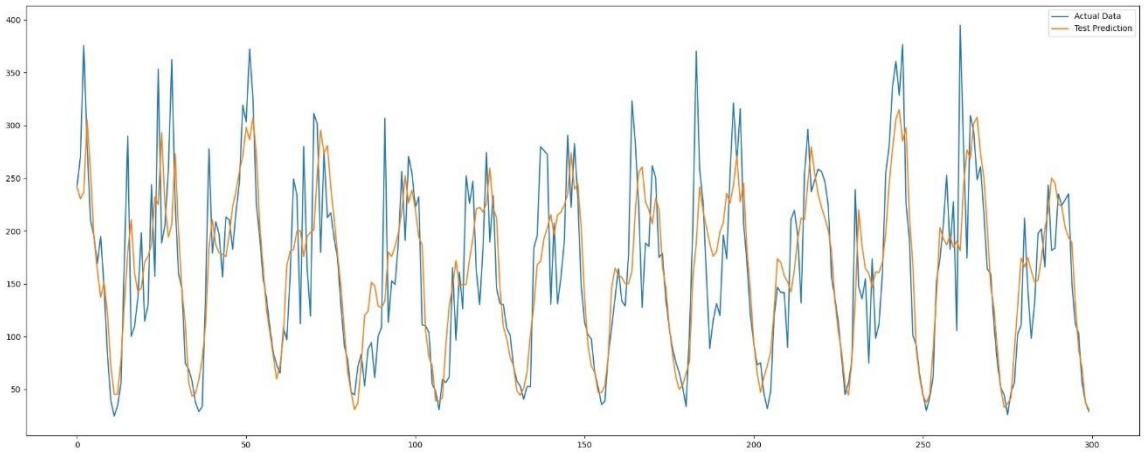

Figura 6.9: *Fragmento de las predicciones del conjunto de test del modelo LSTM_stacked_3*

Como se ha comentado anteriormente, la variabilidad de la serie temporal tiene una tendencia creciente por lo que es probable que el error también lo haga. Este fenómeno puede provocar que la evaluación del modelo mediante las métricas MAE y RSME no sean las más apropiadas ni justas. Por esa razón, se han analizado los errores absolutos y los relativos mediante el MAPE (Mean Absolute Percentage Error) de una forma más descriptiva. Para calcularlos se ha utilizado el notebook *error_assessement.ipynb.* En la Tabla 6.2 se muestran los resultados obtenidos tanto para el conjunto de entrenamiento como el de test.

Tabla 6.2: *Resultados de los errores absolutos y del MAPE*

Estadísticas	Error Abs Train	Error Abs Test	MAPE Train	MAPE Test
Min	0.002	0.093	0	0
Q1	9.174	13.058	6.9%	8.5%
Median	20.988	30.369	17.6%	20.9%
Mean	30.174	39.809	Inf	Inf
Q3	43.435	57.238	40%	45.1%
Max	332.234	225.939	Inf	Inf

En referencia al error absoluto, se observa como la media del error de entrenamiento coincide con el valor de MAE que se calculó en la fase de validación mientras que la media del conjunto de test es ligeramente superior al MAE del conjunto de validación. Este hecho refuerza la hipótesis expuesta anteriormente sobre el aumento del error a lo largo de la serie temporal ya que el conjunto de datos de validación es precedente en el tiempo al conjunto de test.

Por otro lado, si analizamos el MAPE se puede observar como el 25 % de las muestras tienen menos de un 7% de error en el entrenamiento y menos de un 8.5% en el test. Además, la media de error del conjunto de test es del 17.6% y la de test del 20.9%. En referencia a los infinitos que salen, es debido a que existen varias franjas horarias con 0 demanda eléctrica.

Este nuevo enfoque parece señalar que el modelo predice mejor de lo que indicaban las métricas calculadas en la fase de validación, por lo que es evidente que la tendencia creciente de la variabilidad en la serie temporal afecta a los errores. Además, el modelo presenta dificultades al predecir los valores extremos superiores, donde el error absoluto es significativamente alto, lo que penaliza mucho el rendimiento general del modelo.

En cuanto a los resultados de las predicciones a futuro, en la *Figura 6.10* se puede observar las predicciones a futuro de las primeras 24h del conjunto de test. Obviamente, las predicciones son mucho menos precisas que cuando el horizonte temporal era 1. Sin embargo, si se calcula el error relativo total de las 24h, los errores positivos y negativos se compensan y el modelo termina generando un solo un 7.59% de error.

En la *Figura 6.11* se muestran las predicciones de 7 días consecutivos manteniendo el horizonte temporal a 24h. A simple vista se puede ver que las predicciones tienen una forma bastante similar a lo largo de los días por lo que el modelo no es capaz de capturar las fluctuaciones horarias.

En el notebook *test_assessement,* también se calculó el error relativo promedio de todo el conjunto de test en clústeres de 24h, cubriendo un período de 30 días. Este error relativo promedio fue del 20.65%. Cabe remarcar que, un día en particular, con un error relativo de 228.94%, afectó significativamente este promedio.

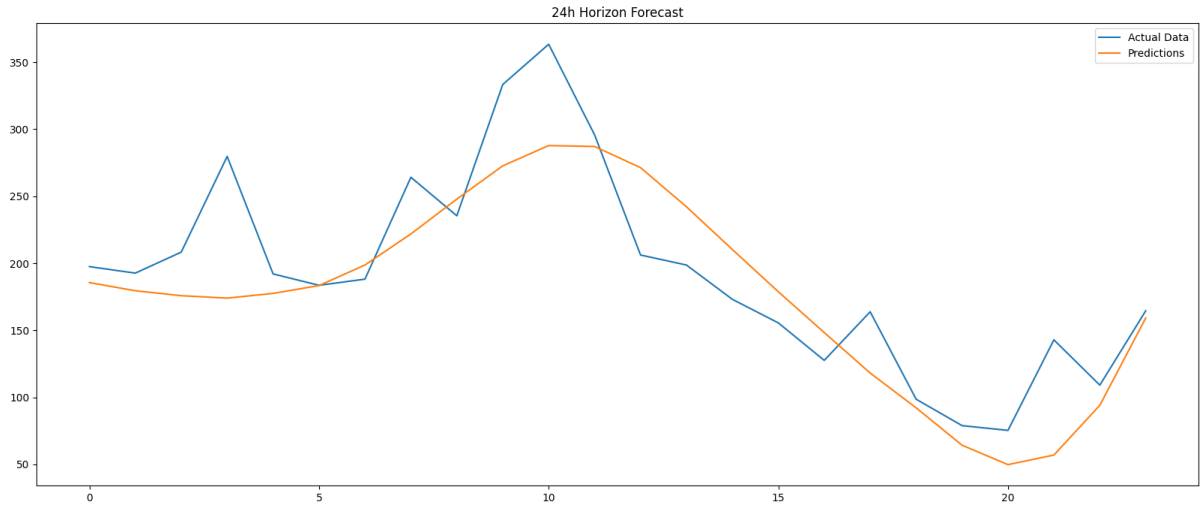

Figura 6.10: *Predicción a futuro de 24h con un horizonte de 24h*

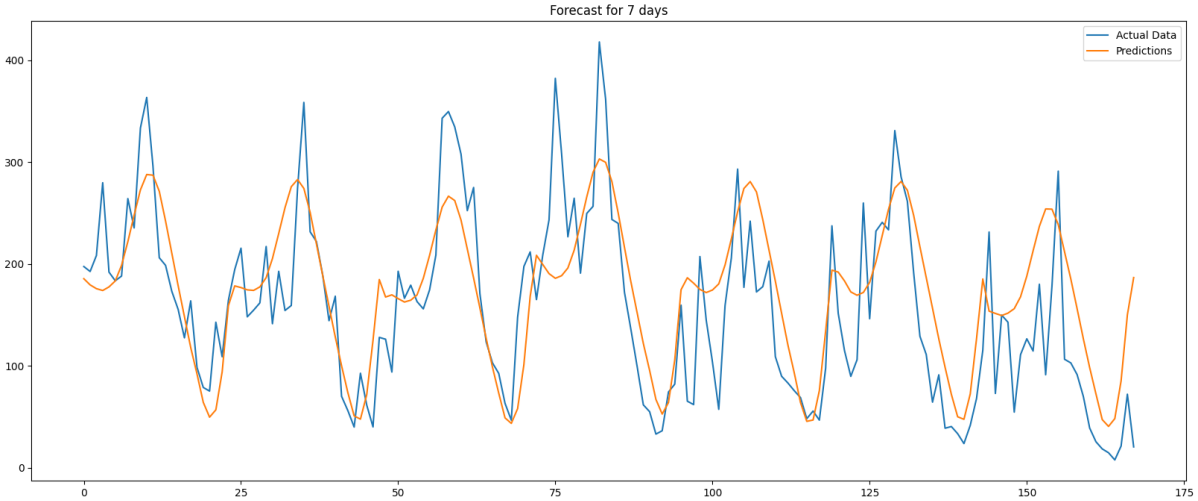

Figura 6.11: *Predicciones a futuro de 7 días con un horizonte de 24h*

Finalmente, en la *Figura 6.12*, se representan los resultados de la comparativa de las predicciones de todos los modelos utilizados en el proyecto con un batch size de 32 y una window size de 96. Las tres graficas muestran un periodo de 72h consecutivas.

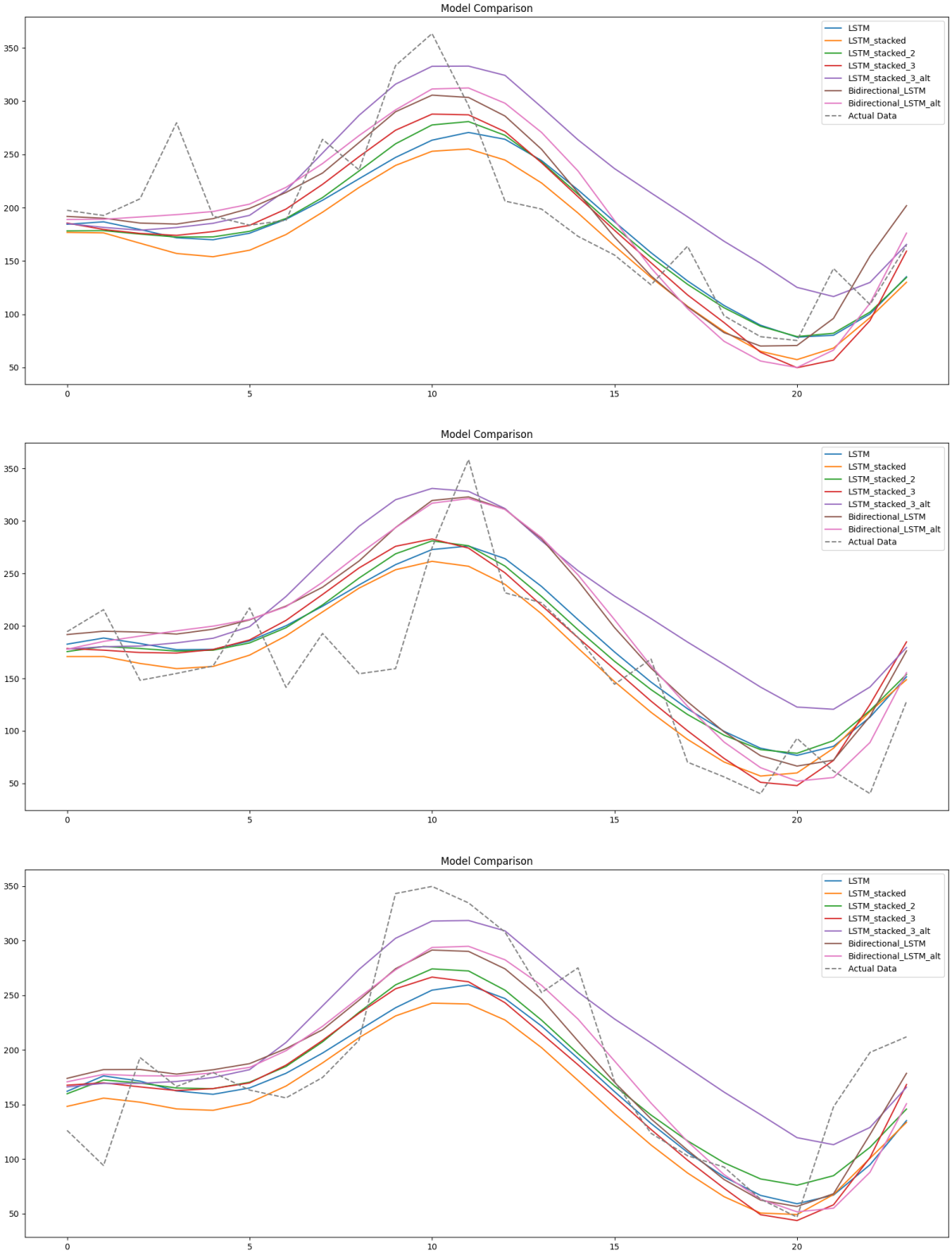

***Figura 6.12**: Comparación de las predicciones de los modelos en horizontes de 24h*

7 Conclusiones

El presente trabajo de fin de master ha abordado el desafío de predecir la demanda eléctrica mediante el uso de técnicas de Deep Learning como son las redes neuronales recurrentes LSTM. A lo largo del estudio, se han implementado y evaluado diferentes modelos con el objetivo de mejorar la precisión de las predicciones. La fase de entrenamiento ha sido satisfactoria ya que, en todos los modelos, los errores de validación se han reducido a la vez que los de entrenamiento sin llegar al overfitting.

En referencia a la fase de validación, la mayoría de modelos han tenido unas métricas similares que variaban 5 puntos aproximadamente entre ellos (teniendo como referencia que la escala de la serie temporal es de 0 – 500 unidades). Se percibió una ligera mejora del rendimiento en los modelos más complejos y con un tamaño de lote más pequeño, sin embargo, parece que todo el margen de mejora que se podía obtener utilizando diferentes arquitecturas y configuraciones de hiperparámetros ya ha sido agotada.

En cuanto a la fase de test, se ha observado que el modelo consigue capturar los patrones a nivel diario, exceptuando los valores más extremos, pero las fluctuaciones horarias siguen escapándose. Sin embargo, como punto a favor del modelo, se ha observado que el error porcentual absoluto medio (MAPE) del conjunto de test es relativamente bajo, indicando así que el modelo es capaz de predecir bien, pero se ve penalizado considerablemente por los valores extremos y por la tendencia creciente de la variabilidad de la serie temporal.

Finalmente, se realizaron predicciones a futuro con un horizonte temporal de 1 h y de 24 h, siendo la primera más precisa. También se analizaron los errores relativos de las predicciones a futuro y se observó que, a pesar de la imprecisión horaria de las predicciones, en un periodo de 24h, los errores positivos y negativos se compensaban y el modelo era capaz de predecir con un error relativo medio del 20.65% para todo el conjunto de test.

Considerando todos los puntos anteriores, parece ser que el modelo es capaz de predecir con relativamente precisión la demanda siempre y cuando el enfoque sea la predicción diaria de electricidad.

Debido a al tiempo limitado para realizar este proyecto no se pudo refinar más el modelo, pero me gustaría proponer una serie de sugerencias para un trabajo futuro:

- Realizar un estudio utilizando redes neuronales recurrentes pero multivariables. Se podría considerar el mes, el día de la semana, información meteorológica…
- Recopilar más data para tener al menos 2 años enteros y mejorar el entrenamiento del modelo.
- Aprovechando que el dataset tiene muestras de distintas localizaciones, se podría realizar un estudio especifico de algunas de ellas para observar si la variabilidad de la serie disminuye y el modelo puede capturar los patrones de forma más efectiva.

8 Referencias

[1] I. E. A. (IEA), «Global EV Outlook 2020: Entering the decade of electric drive?,» IEA, París, 2020.

[2] «GeeksforGeeks,» 10 June 2024. [En línea]. Available: https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/.

[3] K. L. E. &. K. J. Baek, «A dataset for multi-faceted analysis of electric vehicle charging transactions,» *Sci Data ,* nº https://doi.org/10.1038/s41597-024-02942-9, 2024.