

Transformers para series de tiempo

 render  nbviewer  Open in Colab

トラック *Instalaciones:* トラック

Este notebook utiliza [Poetry](#) para la gestión de dependencias. Primero instala Poetry siguiendo las instrucciones de su [documentación oficial](#). Luego ejecuta el siguiente comando para instalar las dependencias necesarias y activar el entorno virtual:

- Bash:

```
poetry install  
eval $(poetry env activate)
```

- PowerShell:

```
poetry install  
Invoke-Expression (poetry env activate)
```

手 *Importaciones:* 手

```
In [ ]: import numpy as np  
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
from statsmodels.tsa.stattools import adfuller  
  
import keras  
from keras import Input, Model  
from keras.layers import (  
    LayerNormalization,  
    MultiHeadAttention,  
    Dropout,  
    LayerNormalization,  
    Conv1D,  
    Dense,  
    GlobalAveragePooling1D,  
    Layer,  
    Embedding,  
)  
from keras.callbacks import EarlyStopping, History  
from keras.losses import MeanSquaredError  
from keras.optimizers import Adam  
from keras.metrics import MeanAbsoluteError
```

```

from sklearn import metrics

from typing import Tuple

from pprint import pprint

import torch
import torch.nn as nn
from transformers import TimesFmModelForPrediction, PatchTSTConfig, PatchTSTForPred
from sklearn.preprocessing import StandardScaler
from torch.utils.data import Dataset, DataLoader

from sklearn.metrics import mean_squared_error, mean_absolute_error, root_mean_squa

```

🔧 *Configuraciones:* 🔧

In [2]:

```

WINDOW_SIZE = 5
EMBEDDING_DIM = 8
DEVICE = 'cuda' if torch.cuda.is_available() else 'cpu'

```



Figura 1: Un científico analizando series de tiempo de mercados financieros utilizando redes neuronales

🌟 Datos del proyecto: 🌟

Subtítulo Transformers en series de tiempo - Análisis de series temporales 2 - FIUBA

Descripción Estudio sobre la aplicación de *Transformers* en series de tiempo.

Integrantes - Bruno Masoller (brunomaso1@gmail.com)- Fabricio Denardi (denardifabricio@gmail.com) - Francisco Rassi (franciscorassi@gmail.com)

Tabla de Contenido

- 1. Introducción a los *Transformers*
- 2. Transformers para series de tiempo
- 2.1. Embeddings y positional encoding
- 3. Caso práctico
 - 3.1. Transformer base
 - 3.2. Transformer + positional encoding
- 4. Estado del arte
 - 4.1 Caso de uso: TimesFM
- 5. Resultados
 - 5.1. Conclusiones
 - 5.2. Mejoras y futuras líneas de investigación
- 6. Referencias

1. Introducción a los *Transformers*

Los *Transformers* son una arquitectura de red neuronal introducida en el artículo "Attention is All You Need" por Vaswani et al. en 2017. Originalmente diseñados para tareas de procesamiento de lenguaje natural (NLP), los *Transformers* han demostrado ser altamente efectivos en una variedad de tareas, incluyendo traducción automática, generación de texto y análisis de sentimientos. Esta arquitectura se basa en el mecanismo de atención, que permite a la red enfocarse en diferentes partes de la entrada de manera dinámica, lo que mejora la capacidad de modelar relaciones a largo plazo en los datos. Una representación visual del artículo puede verse en la siguiente figura:

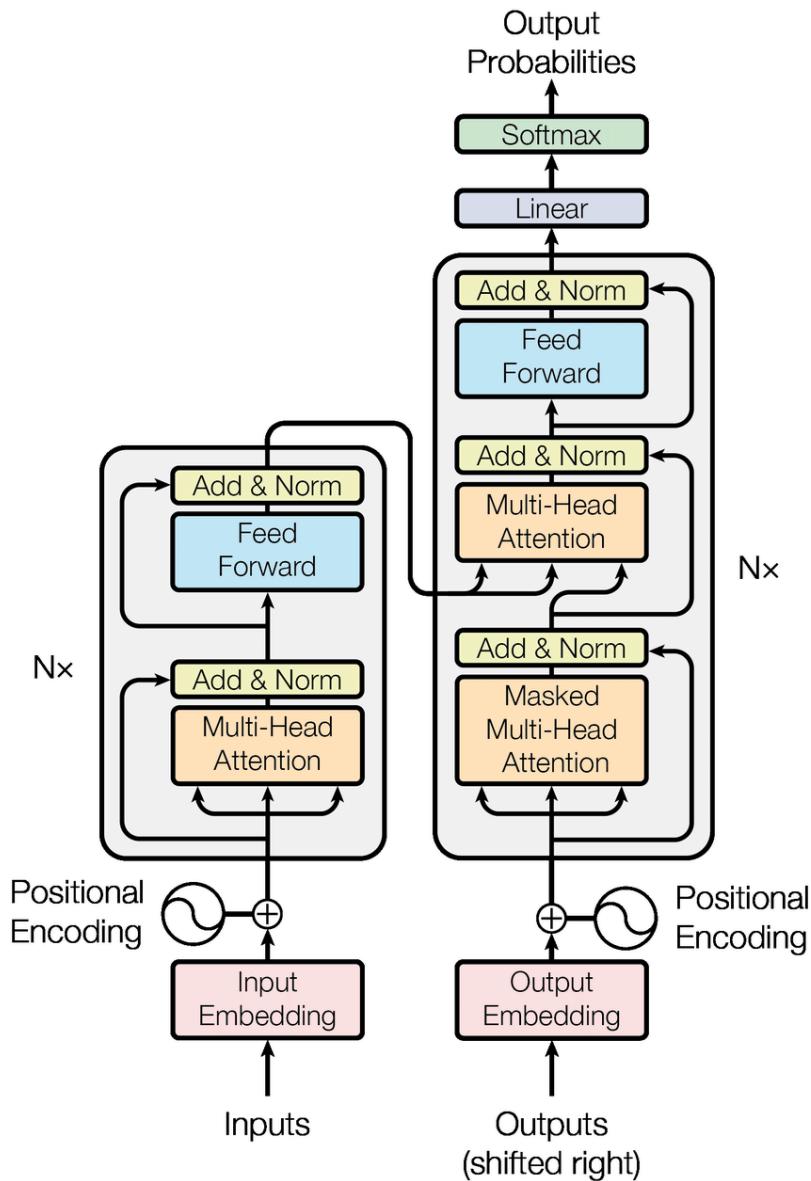


Figura: Arquitectura del Transformer, tomada de "Attention is All You Need" (Vaswani et al., 2017).

Más allá de su éxito en NLP, los *Transformers* han sido adaptados para otras áreas, como visión por computadora, en el artículo "An Image is Worth 16x16 Words" por Dosovitskiy et al. en 2020, donde se introdujo el Vision Transformer (ViT); en el modelado espacio-temporal, introducido en el artículo "A 3d high-resolution model for fast and accurate global weather forecast" por Bi et al. en 2022; o en reconocimiento de voz, como se describe en el artículo "Speech-Transformer: a no-recurrence sequence-to-sequence model for speech recognition" por Dong et al. en 2018. En series de tiempo, también se han explorado diversas adaptaciones de *Transformers*.

El objetivo de este trabajo es explorar el uso de *Transformers* en series de tiempo, centrándonos en su aplicación para la predicción de series temporales financieras. A través de un conjunto de datos clásico, como lo es el precio del *Bitcoin*, analizaremos el rendimiento de varios tipos de *Transformers*.

💡 **Más información:** Esta es una breve introducción, si deseas más información puedes mirar este artículo donde se explica con profundidad varios conceptos: <https://arxiv.org/pdf/2304.10557.pdf>

2. Transformers para series de tiempo

La predicción de series de tiempo enfrenta varios desafíos debido a las características propias de los datos. Usualmente, estos desafíos se asocian con conceptos de estacionariedad, linearidad y naturaleza caótica. Enfoques tradicionales basados en aprendizaje profundo como RNN (*Recurrent Neural Networks*), LSTMs (*Long Short-Term Memory*) o GRU (*Gated Recurrent Units*) procesan los datos de forma secuencial pero son inefficientes para secuencias largas.

Para superar estos desafíos, una línea de investigación moderna se basa en la aplicación de los *Transformers* a estas secuencias de datos, focalizándose en las ventajas que presentan estas arquitecturas en el procesamiento de datos. Entre las principales ventajas se encuentran:

- **Paralelización:** A diferencia de las RNN, los *Transformers* permiten el procesamiento paralelo de secuencias, lo que acelera el entrenamiento y la inferencia.
- **Atención:** El mecanismo de atención permite a los *Transformers* enfocarse en diferentes partes de la secuencia de entrada, capturando relaciones a largo plazo y mejorando la capacidad de modelar dependencias complejas.
- **Escalabilidad:** Los *Transformers* son altamente escalables y pueden manejar secuencias de longitud variable, lo que los hace adecuados para una amplia gama de tareas de series de tiempo.

En base al artículo "A Survey on Transformers for Time Series Forecasting" de Wu et al. (2023), una posible taxonomía para clasificar a los *Transformers* para series de tiempo se puede dividir en dos categorías principales: según su arquitectura y según su aplicación, como puede verse en la siguiente figura:

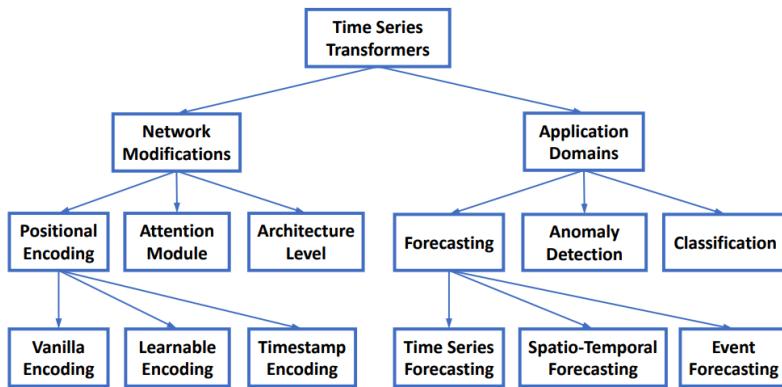


Figura: Taxonomía de Transformers para series de tiempo según Wu et al. (2023).

Un resumen de esta clasificación es el siguiente:

- Clasificación según modificaciones en la arquitectura:
 - Modificaciones en "Positional Encoding".
 - Vanilla positional encoding.
 - Learnable positional encoding.
 - Timestamp encoding
 - Modificaciones (optimizaciones) en el módulo de atención.
 - Introducción de sesgo espacial (LogTrans y Pyraformer)
 - Exploración de las propiedades low-rank de la matriz de atención (Informer y FEDformer)
 - Innovación en la arquitectura de la atención
 - Arquitecturas jerárquicas
- Clasificación según el tipo de aplicación.
 - Predicción
 - Predicción de series temporales (Informer, etc.)
 - Predicciones temporales-espaciales
 - Predicciones de eventos.
 - Detección de anomalías
 - Clasificación

Varias variantes de *Transformers* han sido propuestas para abordar estos desafíos, cada una con sus propias innovaciones y optimizaciones. Por ejemplo, según la tarea de predicción, se puede nombrar los artículos de "Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting" por Li et al. (2019) o "Informer: Beyond efficient transformer for long sequence time-series forecasting" por Zhou et al. (2021), que introducen optimizaciones en el módulo de atención para mejorar la eficiencia y la precisión en la predicción de series temporales. Así mismo, para la detección de anomalías, se puede mencionar el artículo "TranAD: Deep transformer networks for anomaly detection in multivariate time series" por Zheng et al. (2021) o el artículo "Voice2Series: Reprogramming acoustic models for time series classification" por Yang et al. (2021) para la tarea de clasificación.

Sin embargo, a pesar de los avances en el uso de *Transformers* para series de tiempo, aún existen desafíos y limitaciones, especialmente en como efectivamente modelar series temporales complejas para capturar la estacionalidad, como se nombra en el artículo "Robust time series analysis and applications: An industrial perspective" por Wen et al. (2022).

En base a esa curiosidad, en las siguientes secciones se explorará el uso de *Transformers* para series de tiempo, centrándose en su aplicación para la predicción de series temporales financieras. A través de un conjunto de datos clásico, como lo es el precio del *Bitcoin*, analizaremos el rendimiento de varios tipos de *Transformers*.

2.1. Embeddings y Positional Encoding

Los *Transformers* proveen una poderosa arquitectura para el procesamiento de una amplia gama de modalidades de datos, como textos e imágenes (explicados anteriormente). En todos estos casos, los datos en "crudo" (raw data) deben primeramente ser "tokenizados" para que se conviertan en una secuencia de vectores que luego pueden ser procesados por el modelo. Debido a que los *Transformers* son invariantes a las permutaciones de los datos, es necesario incorporar información de orden en la secuencia de entrada. Para esto, se utiliza el "positional encoding", que agrega información de posición a cada token en la secuencia.

Existen varias formas de "tokenizar" las series de tiempo, entre las cuales se encuentran utilizar multi-series para representar mejor la información semántica, que al utilizar varios canales, mejora la información semántica local, como se comenta en el paper "A time series is worth 64 words: Long-term forecasting with transformers" por Nie et al. (2023). Otras formas incluyen características (handcrafted features) adicionales, como feriados o días especiales, como el modelo Informer.

En la siguiente sección se muestra un ejemplo para un caso "sin positional encoding" y otro "con positional encoding" como estudio.

3. Caso práctico

Esta sección tiene como objetivo presentar un caso práctico de aplicación de *Transformers* a series de tiempo. El enfoque principal se centra en explorar la construcción de modelos de predicción de series temporales utilizando la arquitectura *Transformer*, construidos de forma "manual" utilizando *Keras*.

Se divide en dos partes, por un lado, se realiza la implementación de un modelo *Transformer* desde cero, en donde las entradas son simplemente series de tiempo univariadas. Por otro lado, se explora y analiza una codificación posicional alternativa, que permite incorporar información adicional al modelo.

Las capas utilizadas en este caso práctico (como sus enlaces a la documentación oficial) son:

- MultiHeadAttention
- PositionalEncoding
- Conv1D
- Dropout
- Dense
- LayerNormalization
- Input
- Flatten

Para este ejemplo, se utilizará el mismo conjunto de datos que se utilizó en la primera versión de la materia (precio del *Bitcoin* obtenido desde Binance), el cual se encuentra disponible en: <https://github.com/brunomaso1/uba-mia/tree/mia-ast1/mia-ast1/Trabajo%20final/datasets>.

Las métricas obtenidas en ese trabajo fueron las siguientes:

```
In [3]: df_metrics = pd.read_csv('resources/previous_metrics_comparison.csv')
df_metrics
```

```
Out[3]:
```

	model	mae	mape	rmse
0	Naive	1735.726500	1.888647	2434.504211
1	ARIMA(1,1,35)	1891.662281	2.063588	2546.257391
2	ARIMA(0,1,0)	1735.726500	1.894494	2434.504211
3	Prophet	2582.551322	2.871448	3621.703076
4	XGBoost	4319.031819	4.609317	5555.304516
5	LSTM	2432.965561	2.645095	3192.781096

Cargamos el conjunto de datos:

```
In [4]: df = pd.read_pickle('resources/BTCUSDT_1D.pkl')
df.head()
```

Out[4]:

	Open	High	Low	Close	Volume	Close Time	Quote Asset Volume	N
Open Time								
2024-03-01	61130.99	63114.23	60777.00	62387.90	47737.93473	2024-03-01 23:59:59.999	2.956537e+09	19
2024-03-02	62387.90	62433.19	61561.12	61987.28	25534.73659	2024-03-02 23:59:59.999	1.582567e+09	16
2024-03-03	61987.28	63231.88	61320.00	63113.97	28994.90903	2024-03-03 23:59:59.999	1.804536e+09	19
2024-03-04	63113.97	68499.00	62300.00	68245.71	84835.16005	2024-03-04 23:59:59.999	5.568878e+09	38
2024-03-05	68245.71	69000.00	59005.00	63724.01	132696.78130	2024-03-05 23:59:59.999	8.674527e+09	53



En este caso, nos quedamos solo con la columna de "Close" del conjunto de datos, que representa el precio de cierre de BTC:

In [5]:

```
df = df[['Close']]
df.head()
```

Out[5]:

Close

Open Time	Close
2024-03-01	62387.90
2024-03-02	61987.28
2024-03-03	63113.97
2024-03-04	68245.71
2024-03-05	63724.01

Guardamos una copia:

In [6]:

```
df.to_csv('resources/BTCUSDT_1D.csv', index=True)
```

Inicialmente, realizamos un test de estacionaridad, utilizando la prueba de Dickey-Fuller aumentada (ADF). Para ello, generamos una función auxiliar que realice el test:

In [7]:

```
def test_stationarity(timeseries: pd.DataFrame) -> None:
    """
    Realiza la prueba de Dickey-Fuller aumentada (ADF) para evaluar la estacionarid
```

```

Args:
    timeseries (pd.DataFrame): Serie temporal a evaluar, puede ser un DataFrame

Prints:
    Estadístico ADF, valor p y conclusión sobre la estacionariedad de la serie.
"""

adf_result = adfuller(timeseries)
print("ADF Statistic:", adf_result[0])
print("p-value:", adf_result[1])

if adf_result[1] <= 0.05:
    print("La serie es estacionaria (rechazamos H0)")
else:
    print("La serie no es estacionaria (no rechazamos H0)")

```

Realizamos el test:

In [8]: `test_stationarity(df)`

```

ADF Statistic: -1.420856101366877
p-value: 0.5722376131152292
La serie no es estacionaria (no rechazamos H0)

```

Nota: Se podría realizar otras pruebas como KPSS, no linealidad (BDS), normalidad (Shapiro-Wilks), etc. pero no es el objetivo de este trabajo. Por más información sobre el conjunto de datos, se puede acceder en el siguiente enlace: <https://github.com/brunomaso1/uba-mia/blob/mia-ast1/mia-ast1/Trabajo%20final/tp-final.ipynb>

Dividimos el conjunto de datos en entrenamiento y test (también convertimos los datos en listas):

In [9]: `train_size = int(len(df) * 0.9)
df_train, df_test = df[:train_size].values, df[train_size:].values

spots_train, spots_test = df_train.tolist(), df_test.tolist()

print(f"Train size: {len(spots_train)}")
print(f"Test size: {len(spots_test)}")`

```

Train size: 356
Test size: 40

```

Definimos una función auxiliar para crear las secuencias de datos:

In [10]: `def to_sequences(seq_size: int, obs: list) -> Tuple[np.ndarray, np.ndarray]:
"""

Genera secuencias de longitud fija y sus correspondientes etiquetas para problema

Args:
 seq_size (int): Tamaño de la ventana o longitud de cada secuencia.
 obs (list): Lista de observaciones (puede ser una lista de listas o valores`

```

Returns:
    tuple[np.ndarray, np.ndarray]:
        - x: Array de secuencias de entrada de tamaño (n_samples, seq_size, ...)
        - y: Array de valores objetivo correspondientes a cada secuencia.
    """
    x = [np.array(obs[i : i + seq_size]) for i in range(len(obs) - seq_size)]
    y = [obs[i + seq_size] for i in range(len(obs) - seq_size)]
    return np.array(x), np.array(y)

```

Creamos las secuencias de datos:

```
In [11]: X_train, y_train = to_sequences(WINDOW_SIZE, spots_train)
X_test, y_test = to_sequences(WINDOW_SIZE, spots_test)
print(f"X_train shape: {X_train.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_test shape: {y_test.shape}")
```

```
X_train shape: (351, 5, 1)
y_train shape: (351, 1)
X_test shape: (35, 5, 1)
y_test shape: (35, 1)
```

3.1. Transformer base

Para la construcción del modelo, se crean dos funciones auxiliares, una que crea el encoder y otra que construye el modelo.

El modelo se construye utilizando la API funcional de `Keras`, donde se define un encoder que utiliza capas de atención y capas densas para procesar las secuencias de datos. El bloque `transformer_encoder` cuenta con:

- Capas de normalización de datos (`LayerNormalization`).
- Capas de atención multi-cabeza (`MultiHeadAttention`).
- Sub-red feed-forward con capas convolucionales (`Conv1D`).
- Capa de dropout para regularización (`Dropout`).
- Conexiones residuales para mejorar el flujo de gradientes.

 **MultiHeadAttention:** Esta capa es fundamental en las arquitecturas de *Transformers* y se utiliza para permitir que el modelo "atienda" o preste atención a diferentes partes de una secuencia de entrada. En lugar de una sola cabeza de atención, utiliza múltiples "cabezas" de atención que aprenden a enfocar en diferentes aspectos de la información de entrada de manera paralela (en este caso, la serie temporal). Luego, las salidas de estas múltiples cabezas se concatenan y se proyectan linealmente. Esto permite que el modelo capture relaciones complejas y de largo alcance dentro de la secuencia, mejorando su capacidad para comprender el contexto.

- **Usos comunes:** En el PNL, es fundamental en modelos de transformadores para traducción, resumen, clasificación de texto, respuesta a preguntas y generación de texto, al entender relaciones entre palabras distantes. También se aplica en visión por computadora para tareas como clasificación de imágenes, detección de objetos y segmentación, al capturar dependencias espaciales en imágenes.
- **Dimensiones de entrada/salida:** Usualmente, la capa toma tres entradas: query, key y value. Cada una con forma `(batch_size, sequence_length, embedding_dim)`. En la auto-atención, estas tres entradas suelen ser el mismo tensor o transformaciones lineales del mismo. Como salida, se tiene la misma forma que la query de entrada: `(batch_size, query_sequence_length, embedding_dim)`

💡 **Conv1D:** La capa Conv1D realiza una operación de convolución unidimensional. Esto significa que aplica un filtro convolucional deslizante a lo largo de una única dimensión de la entrada. Cada filtro se "desliza" sobre la secuencia de entrada, calculando el producto escalar entre los valores del filtro y los valores correspondientes en la entrada.

- **Usos comunes:** Es muy utilizada en PNL para extraer características de secuencias de texto (por ejemplo, reconocer n-gramas o patrones de palabras). También se aplica en el procesamiento de señales de tiempo (como audio o datos de sensores) para detectar patrones temporales.
- **Dimensiones de entrada/salida:** Típicamente, toma una entrada con forma `(batch_size, steps, features)` y produce una salida con forma similar, pero con las steps reducidas según el tamaño del filtro y el stride, y el número de features determinado por la cantidad de filtros.

💡 **GlobalAveragePooling1D:** Esta capa es una forma de reducción de dimensionalidad. Para cada característica en la entrada, calcula el promedio de todos los valores a lo largo de la dimensión de la secuencia. En otras palabras, toma una secuencia de vectores y la convierte en un solo vector promediando cada característica de forma independiente a lo largo del eje temporal.

- **Usos comunes:** A menudo se utiliza después de una capa convolucional (como Conv1D) para aplanar las características extraídas y prepararlas para una capa densa (totalmente conectada). Sirve como una forma de resumir la información más importante de la secuencia sin perder demasiada información clave, y ayuda a reducir el número de parámetros del modelo, lo que puede prevenir el sobreajuste.
- **Dimensiones de entrada/salida:** Si la entrada es `(batch_size, steps, features)`, la salida será `(batch_size, features)`, donde

`features` es el número de canales de características.

Función `transformer_encoder` para crear el encoder:

```
In [12]: def transformer_block(
    inputs: keras.layers, head_size: int, num_heads: int, ff_dim: int, dropout: float
) -> keras.layers:
    """
    TODO: Mejorar esta función cambiando a una Layer de Keras personalizada + Test
    Construye un bloque codificador tipo Transformer.

    Args:
        inputs (keras.Layer): Capa de entrada al encoder.
        head_size (int): Dimensión de cada cabeza de atención.
        num_heads (int): Número de cabezas de atención.
        ff_dim (int): Dimensión de la red feed-forward interna.
        dropout (float, optional): Tasa de dropout. Por defecto 0.

    Returns:
        keras.Layer: Capa de salida del bloque codificador Transformer.
    """
    # Normalización de la entrada
    x = LayerNormalization(epsilon=1e-6)(inputs) # Shape: (batch_size, seq_len, features)

    # Atención multi-cabeza
    x = MultiHeadAttention(key_dim=head_size, num_heads=num_heads, dropout=dropout)

    # Conexión residual
    x = Dropout(dropout)(x) # Shape: (batch_size, seq_len, features) -> (batch_size, seq_len, features)
    res = x + inputs # Shape: (batch_size, seq_len, features) -> (batch_size, seq_len, features)

    # Normalización de la salida
    x = LayerNormalization(epsilon=1e-6)(res) # Shape: (batch_size, seq_len, features)

    # Capa densa feed-forward
    x = Conv1D(filters=ff_dim, kernel_size=1, activation="relu")(x) # Shape: (batch_size, seq_len, ff_dim)
    x = Dropout(dropout)(x) # Shape: (batch_size, seq_len, ff_dim) -> (batch_size, seq_len, ff_dim)

    return Conv1D(filters=inputs.shape[-1], kernel_size=1)(x) + res # Shape: (batch_size, seq_len, features)
```

Función `build_model` para construir el modelo:

```
In [13]: def build_model(
    input_shape: tuple,
    head_size: int,
    num_heads: int,
    ff_dim: int,
    num_transformer_blocks: int,
    mlp_units: list,
    dropout: float = 0,
    mlp_dropout: float = 0,
) -> keras.Model:
    """
    Construye un modelo basado en bloques Transformer para series temporales.

```

```

Args:
    input_shape (tuple): Forma de la entrada (longitud de la secuencia, número
    head_size (int): Dimensión de cada cabeza de atención.
    num_heads (int): Número de cabezas de atención en MultiHeadAttention.
    ff_dim (int): Dimensión de la red feed-forward interna de cada bloque Trans
    num_transformer_blocks (int): Número de bloques codificadores Transformer a
    mlp_units (list): Lista con el número de unidades para cada capa densa (MLP
    dropout (float, opcional): Tasa de dropout en los bloques Transformer. Por
    mlp_dropout (float, opcional): Tasa de dropout en las capas MLP. Por defecto

Returns:
    keras.Model: Modelo Keras listo para compilar y entrenar.
"""
inputs = Input(shape=input_shape) # Shape: (seq_len, num_features)
x = inputs

# Apilar bloques de codificador Transformer
for _ in range(num_transformer_blocks):
    x = transformer_block(x, head_size, num_heads, ff_dim, dropout) # Shape: (batch_size, seq_len, num_features)

# Promediar la salida a lo largo de la secuencia
x = GlobalAveragePooling1D(data_format="channels_first")(x) # Shape: (batch_size, num_features)

# Capa MLP
for dim in mlp_units:
    x = Dense(dim, activation="relu")(x) # Shape: (batch_size, num_features) ->
    x = Dropout(mlp_dropout)(x) # Shape: (batch_size, dim) -> (batch_size, dim)

outputs = Dense(1)(x) # Shape: (batch_size, dim) -> (batch_size, 1)

return Model(inputs, outputs)

```

Una vez definidas las funciones auxiliares, se procede a crear el modelo:

```
In [14]: input_shape = X_train.shape[1:]
model = build_model(
    input_shape,
    head_size=256,
    num_heads=4,
    ff_dim=4,
    num_transformer_blocks=4,
    mlp_units=[128],
    dropout=0.25,
    mlp_dropout=0.4
)
```

Mostramos un resumen del modelo:

```
In [15]: model.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 5, 1)	0	-
layer_normalization (LayerNormalizatio...)	(None, 5, 1)	2	input_layer[0][0]
multi_head_attenti... (MultiHeadAttentio...)	(None, 5, 1)	7,169	layer_normalizat... layer_normalizat...
dropout_1 (Dropout)	(None, 5, 1)	0	multi_head_atten...
add (Add)	(None, 5, 1)	0	dropout_1[0][0], input_layer[0][0]
layer_normalizatio... (LayerNormalizatio...)	(None, 5, 1)	2	add[0][0]
conv1d (Conv1D)	(None, 5, 4)	8	layer_normalizat...
dropout_2 (Dropout)	(None, 5, 4)	0	conv1d[0][0]
conv1d 1 (Conv1D)	(None, 5, 1)	5	dropout_2[0][0]

Finalmente, compilamos el modelo y lo entrenamos:

```
In [16]: model.compile(
    loss=MeanSquaredError(), optimizer=Adam(learning_rate=1e-4), metrics=[MeanAbsol
)
history = model.fit(X_train, y_train, validation_split=0.2, epochs=500, batch_size=
```

```

Epoch 1/500
5/5 6s 123ms/step - loss: 2931908352.0000 - mean_absolute_error: 51398.0898 - val_loss: 5286720512.0000 - val_mean_absolute_error: 72658.4375
Epoch 2/500
5/5 0s 31ms/step - loss: 2639362816.0000 - mean_absolute_error: 48537.7188 - val_loss: 4933776896.0000 - val_mean_absolute_error: 70189.2500
Epoch 3/500
5/5 0s 30ms/step - loss: 2355904256.0000 - mean_absolute_error: 45970.3320 - val_loss: 4596278784.0000 - val_mean_absolute_error: 67744.0625
Epoch 4/500
5/5 0s 30ms/step - loss: 2402662400.0000 - mean_absolute_error: 46137.6758 - val_loss: 4269745664.0000 - val_mean_absolute_error: 65291.1602
Epoch 5/500
5/5 0s 30ms/step - loss: 2094848384.0000 - mean_absolute_error: 42966.1719 - val_loss: 3960026624.0000 - val_mean_absolute_error: 62876.1328
Epoch 6/500
5/5 0s 30ms/step - loss: 2153898752.0000 - mean_absolute_error: 43478.9375 - val_loss: 3663840000.0000 - val_mean_absolute_error: 60476.4141
Epoch 7/500
5/5 0s 31ms/step - loss: 1822855040.0000 - mean_absolute_error: 39611.8516 - val_loss: 3384778752.0000 - val_mean_absolute_error: 58124.8398
Epoch 8/500
5/5 0s 30ms/step - loss: 1766975488.0000 - mean_absolute_error: 38791.3711 - val_loss: 3121116416.0000 - val_mean_absolute_error: 55812.0273
Epoch 9/500
5/5 0s 30ms/step - loss: 1664151168.0000 - mean_absolute_error: 37863.9180 - val_loss: 2871860736.0000 - val_mean_absolute_error: 53533.7539
Epoch 10/500
5/5 0s 30ms/step - loss: 1516417664.0000 - mean_absolute_error: 36460.4640 - val_loss: 2620570208.0000 - val_mean_absolute_error: 51207.7205

```

Una vez entrenado el modelo, se procede a verificar los resultados. Para esto, se utilizan varias funciones auxiliares para graficar y evaluar el modelo:

```

In [17]: def smape(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """
    Calcula el Symmetric Mean Absolute Percentage Error (sMAPE) entre los valores reales y los predichos.

    Args:
        y_true (np.ndarray): Valores reales.
        y_pred (np.ndarray): Valores predichos.

    Returns:
        float: El valor de sMAPE expresado en porcentaje.
    """
    return 100 / len(y_true) * np.sum(2 * np.abs(y_pred - y_true)) / (np.abs(y_pred) + np.abs(y_true))

def quantile_loss(q: float, y: np.ndarray, f: np.ndarray) -> np.ndarray:
    """
    Calcula la pérdida de cuantiles (quantile loss) para una predicción dada.

    Args:
        q (float): Cuantil deseado (por ejemplo, 0.5 para la mediana).
        y (np.ndarray): Valores reales.
        f (np.ndarray): Valores predichos.
    """

```

```

    Returns:
        np.ndarray: Pérdida de cuantiles para cada elemento.
    """
    e = y - f
    return np.maximum(q * e, (q - 1) * e)

def calculate_metrics(y_true: np.ndarray, y_pred: np.ndarray) -> dict:
    """
    Calcula varias métricas de evaluación para comparar valores reales y predichos.

    Args:
        y_true (np.ndarray): Valores reales.
        y_pred (np.ndarray): Valores predichos.

    Returns:
        dict: Diccionario con las métricas RMSE, MAPE, MAE, MSE, sMAPE, RRMSE y Quantile Loss.
    """
    return {
        "RMSE": np.sqrt(metrics.mean_squared_error(y_true, y_pred)),
        "MAPE": metrics.mean_absolute_percentage_error(y_true, y_pred) * 100,
        "MAE": metrics.mean_absolute_error(y_true, y_pred),
        "MSE": metrics.mean_squared_error(y_true, y_pred),
        "sMAPE": smape(y_true, y_pred),
        "RRMSE": np.sqrt(metrics.mean_squared_error(y_true, y_pred)) / np.mean(y_true),
        "Quantile Loss": np.mean(quantile_loss(0.5, y_true, y_pred)),
    }

def plot_actual_vs_predicted(y_true: np.ndarray, y_pred: np.ndarray, filename: str):
    """
    Grafica los valores reales frente a los valores predichos y guarda la figura en un archivo.

    Args:
        y_true (np.ndarray): Valores reales.
        y_pred (np.ndarray): Valores predichos.
        filename (str): Nombre del archivo donde se guardará la figura.

    Returns:
        None
    """
    plt.figure(figsize=fig_size, dpi=600)
    plt.plot(y_true, label="Actual", color="blue", linewidth=2)
    plt.plot(y_pred, label="Predicted", color="red", linewidth=2, linestyle="--")
    plt.legend(fontsize="medium", loc="upper left")
    plt.xlabel("Time (Day)", fontsize=16, fontweight="bold")
    plt.ylabel("BTC (BTC/UST)", fontsize=16, fontweight="bold")
    plt.xticks(fontsize=14, fontweight="bold")
    plt.yticks(fontsize=14, fontweight="bold")
    plt.title("Actual vs Predicted", fontsize=16, fontweight="bold")
    plt.savefig(filename, format="jpeg", dpi=600)
    plt.tight_layout()
    plt.show()

```

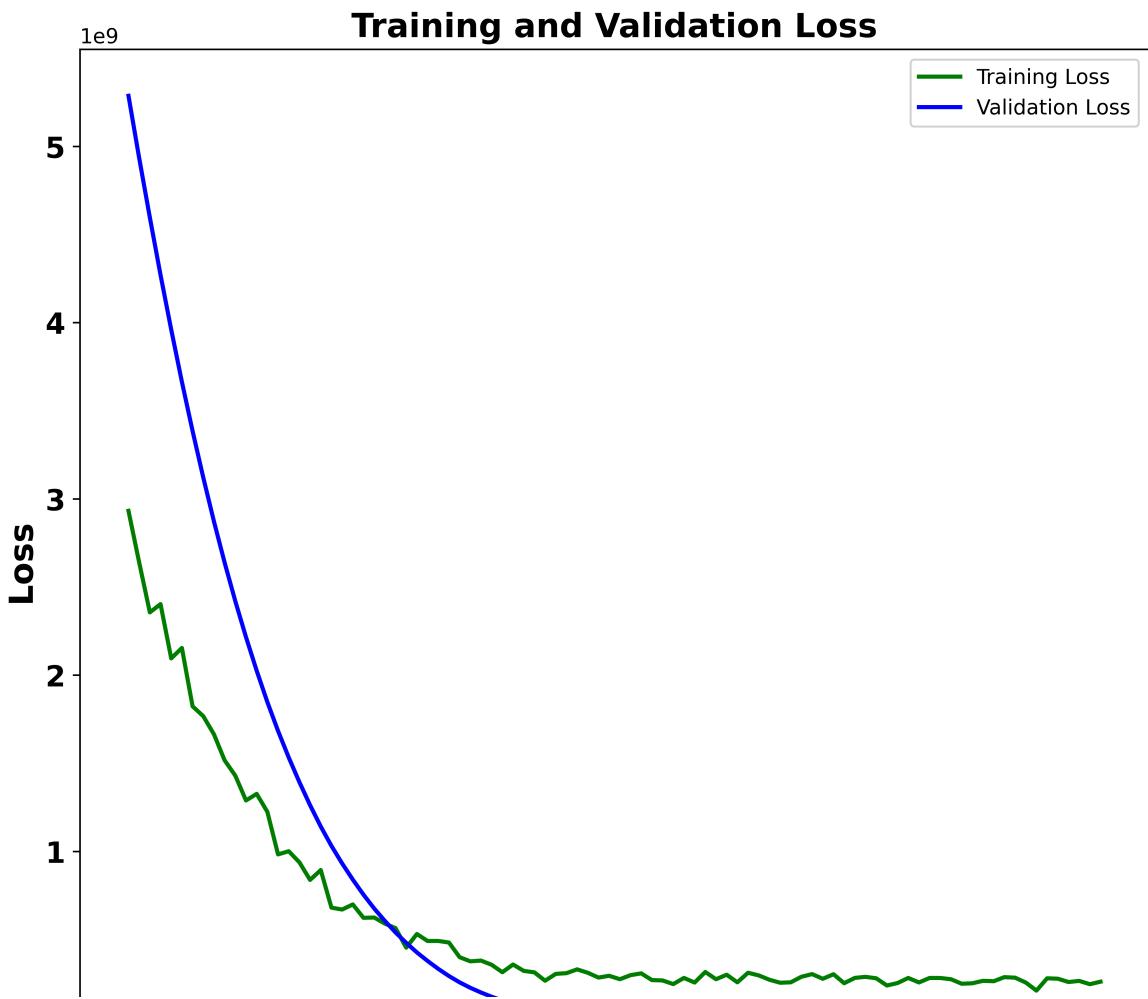
```
def plot_loss(history: History, filename: str, fig_size=(8, 8)) -> None:
    """
    Grafica la pérdida de entrenamiento y validación a lo largo de las épocas y guarda el resultado en un archivo.

    Args:
        history (keras.callbacks.History): Objeto History devuelto por el método fit.
        filename (str): Nombre del archivo donde se guardará la figura.

    Returns:
        None
    """
    plt.figure(figsize=fig_size, dpi=600)
    plt.plot(history.history["loss"], label="Training Loss", color="g", linewidth=2)
    plt.plot(history.history["val_loss"], label="Validation Loss", color="b", linewidth=2)
    plt.legend(fontsize="medium", loc="upper right")
    plt.xlabel("Epochs", fontsize=16, fontweight="bold")
    plt.ylabel("Loss", fontsize=16, fontweight="bold")
    plt.xticks(fontsize=14, fontweight="bold")
    plt.yticks(fontsize=14, fontweight="bold")
    plt.title("Training and Validation Loss", fontsize=16, fontweight="bold")
    plt.savefig(filename, format="jpeg", dpi=600)
    plt.tight_layout()
    plt.show()
```

Mostramos el gráfico de entrenamiento:

```
In [18]: plot_loss(history, "resources/base_model_loss.jpeg", fig_size=(8, 8))
```



Evaluamos el modelo en el conjunto de test:

```
In [19]: test_loss = model.evaluate(X_test, y_test, verbose=1, return_dict=True)
pprint(test_loss)
```

2/2 ━━━━━━━━ 0s 20ms/step - loss: 11054148.0000 - mean_absolute_error: 2
437.5195
{'loss': 11054148.0, 'mean_absolute_error': 2437.51953125}

Calculamos las métricas de evaluación:

```
In [20]: pred_test = model.predict(X_test)
metrics_dict_test = calculate_metrics(y_test, pred_test)
print("\nTest Set Metrics:")
for metric_name, metric_value in metrics_dict_test.items():
    print(f"Score ({metric_name}): {metric_value}")
```

2/2 ————— 0s 245ms/step

Test Set Metrics:

```
Score (RMSE): 3324.778150848964
Score (MAPE): 2.8379590928861047
Score (MAE): 2437.519883928571
Score (MSE): 11054149.752362655
Score (sMAPE): 2.8624316120760143
Score (RRMSE): 3.904685378086401
Score (Quantile Loss): 1218.7599419642854
```

Guardamos las métricas de evaluación:

```
In [21]: df_line = pd.DataFrame({
    "model": "CustTransf",
    "mae": metrics_dict_test["MAE"],
    "mape": metrics_dict_test["MAPE"],
    "rmse": metrics_dict_test["RMSE"]
}, index=[0])

df_metrics = pd.concat([df_metrics, df_line], ignore_index=True)
```

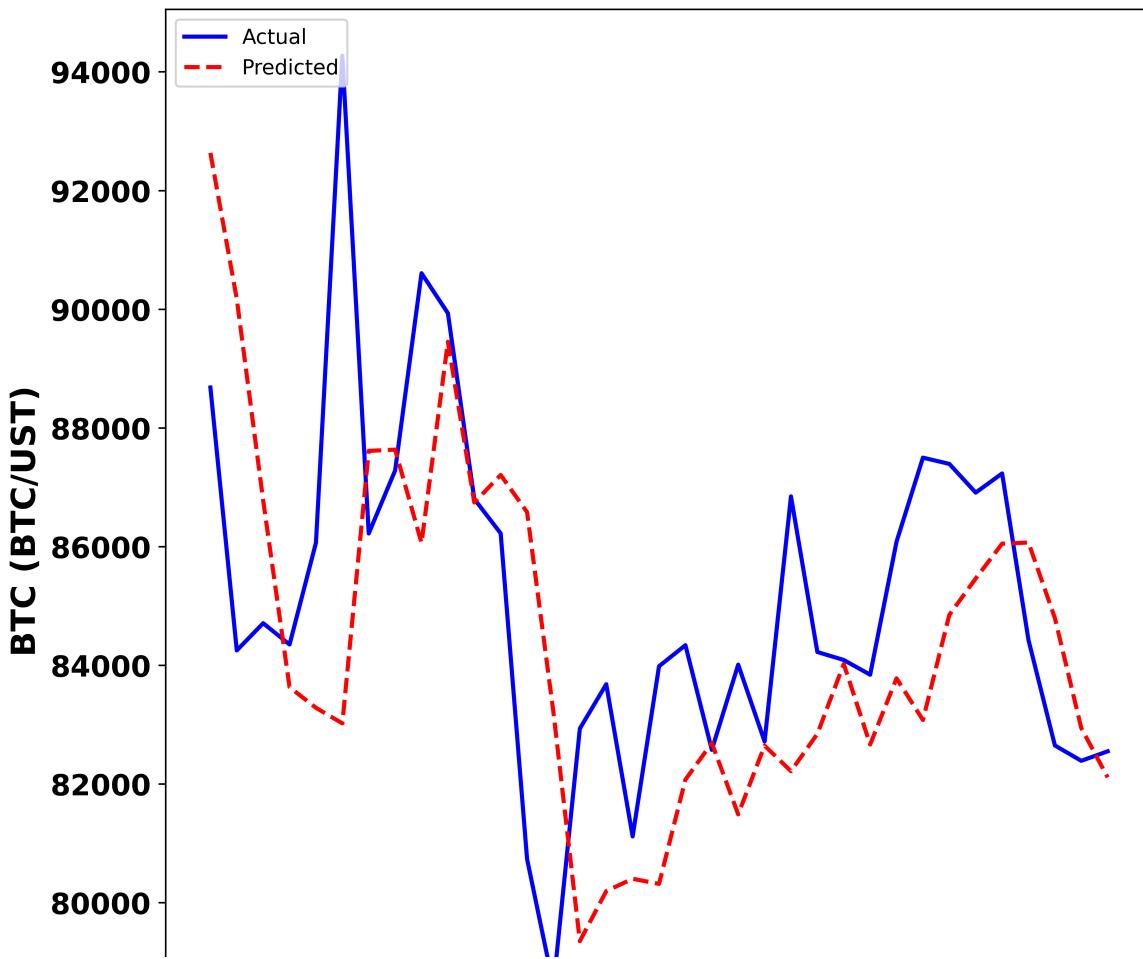
```
In [22]: df_metrics
```

	model	mae	mape	rmse
0	Naive	1735.726500	1.888647	2434.504211
1	ARIMA(1,1,35)	1891.662281	2.063588	2546.257391
2	ARIMA(0,1,0)	1735.726500	1.894494	2434.504211
3	Prophet	2582.551322	2.871448	3621.703076
4	XGBoost	4319.031819	4.609317	5555.304516
5	LSTM	2432.965561	2.645095	3192.781096
6	CustTransf	2437.519884	2.837959	3324.778151

Finalmente, graficamos para observar los resultados de la predicción:

```
In [23]: plot_actual_vs_predicted(y_test, pred_test, "resources/base_model_predict.jpeg")
```

Actual vs Predicted



3.2. Transformer + positional encoding

- *Making new layers and models via subclassing*

El objetivo de esta sección es explorar la incorporación de un *positional encoding* alternativo al modelo *Transformer* previamente construido. Este enfoque busca mejorar la capacidad del modelo para capturar patrones temporales en los datos de series de tiempo. Para ello, se define una nueva clase `PositionalEmbedding` que hereda de `Layer` de Keras. Esta clase implementa un *positional encoding* basado en la codificación de las posiciones de los datos en la secuencia. La idea es que entrada del modelo tenga una representación única que permita que éste aprenda patrones temporales de manera más efectiva.

Ejemplo básico de implementación de un `Layer` de Keras:

```
class Linear(keras.layers.Layer):
    def __init__(self, units=32):
        super().__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
```

```

        shape=(input_shape[-1], self.units),
        initializer="random_normal",
        trainable=True,
    )
    self.b = self.add_weight(
        shape=(self.units,), initializer="random_normal",
        trainable=True
    )

    def call(self, inputs):
        return ops.matmul(inputs, self.w) + self.b

    def get_config(self):
        return {"units": self.units}

```

 *Idea:* A diferencia de NLP, en donde los tokens son discretos y para estos casos se utiliza la capa `Embedding` de Keras, en series de tiempo, los datos son continuos, por lo que para incrustarlos en un espacio de mayor dimensión, probaremos utilizar simplemente una capa `Dense`, en donde el objetivo es que esta capa aprenda a mapear los valores de la serie de tiempo a un espacio dimensional específico (`output_dim`).

Implementación de la clase `PositionalEmbedding`:

```
In [24]: class PositionalEmbedding(Layer):
    """
    Capa de embedding posicional para series de tiempo.

    Args:
        output_dim (int): Dimensión de salida de los embeddings.
        seq_length (int): Longitud de la secuencia de entrada.
        **kwargs: Argumentos adicionales para la clase base Layer.
    """

    def __init__(self, output_dim: int, seq_length: int, **kwargs):
        super().__init__(**kwargs)
        self.output_dim = output_dim
        self.seq_length = seq_length

    def build(self) -> None:
        self.token_embeddings = Dense(self.output_dim, activation=None)
        self.position_embeddings = Embedding(input_dim=self.seq_length, output_dim=)

    def call(self, inputs):
        positions = np.arange(self.seq_length) # Shape: (batch_size, seq_length) ->
        embedded_positions = self.position_embeddings(positions) # Shape: (batch_size, seq_length, output_dim)
        embedded_tokens = self.token_embeddings(inputs) # Shape: (batch_size, seq_length, output_dim)
        # Sumar Los embeddings posicionales y los embeddings de los tokens
        return embedded_tokens + embedded_positions # Shape: (batch_size, seq_length, output_dim)

    def get_config(self) -> dict:
        config = super().get_config()

```

```

        config.update(
            {
                "input_shape": self.input_shape,
                "output_dim": self.output_dim,
            }
        )
    return config

```

Realizamos un test de verificación:

```

inp = Input(shape=(WINDOW_SIZE, 1))
out = PositionalEmbedding(16)(inp)
test_model = Model(inputs=inp, outputs=out)
output = test_model.predict(keras.random.normal(shape=(WINDOW_SIZE,
1)))

```

In [25]:

```

x = keras.random.normal(shape=(WINDOW_SIZE, 1))
y = PositionalEmbedding(output_dim=8, seq_length=WINDOW_SIZE)(x)
y.shape

```

Out[25]: TensorShape([5, 8])

Modificamos la función `build_model` para incorporar la nueva capa

`PositionalEmbedding`, creando una nueva función auxiliar

`build_model_with_positional_embedding`:

In [26]:

```

def build_model_with_embeddings(
    input_shape,
    head_size,
    num_heads,
    ff_dim,
    num_transformer_blocks,
    mlp_units,
    seq_length,
    dropout=0,
    mlp_dropout=0,
    embedding_dim=None,
):
    """
    Construye un modelo Transformer para series de tiempo que incorpora una capa de
    Args:
        input_shape (tuple): Forma de la entrada (longitud de la secuencia, número
        head_size (int): Dimensión de cada cabeza de atención en MultiHeadAttention
        num_heads (int): Número de cabezas de atención.
        ff_dim (int): Dimensión de la red feed-forward interna de cada bloque Trans
        num_transformer_blocks (int): Número de bloques codificadores Transformer a
        mlp_units (list): Lista con el número de unidades para cada capa densa (MLP
        seq_length (int): Longitud de la secuencia de entrada.
        dropout (float, opcional): Tasa de dropout en los bloques Transformer. Por
        mlp_dropout (float, opcional): Tasa de dropout en las capas MLP. Por defecto
        embedding_dim (int, opcional): Dimensión de salida de la capa de embedding.
    Returns:
        Model: El modelo construido.
    """

```

Construye un modelo Transformer para series de tiempo que incorpora una capa de

Args:

- `input_shape` (tuple): Forma de la entrada (longitud de la secuencia, número)
- `head_size` (int): Dimensión de cada cabeza de atención en `MultiHeadAttention`
- `num_heads` (int): Número de cabezas de atención.
- `ff_dim` (int): Dimensión de la red feed-forward interna de cada bloque `Transformer`
- `num_transformer_blocks` (int): Número de bloques codificadores `Transformer` a
- `mlp_units` (list): Lista con el número de unidades para cada capa densa (MLP)
- `seq_length` (int): Longitud de la secuencia de entrada.
- `dropout` (float, opcional): Tasa de dropout en los bloques `Transformer`. Por
- `mlp_dropout` (float, opcional): Tasa de dropout en las capas MLP. Por defecto
- `embedding_dim` (int, opcional): Dimensión de salida de la capa de `embedding`.

Returns:

```

    keras.Model: Modelo Keras listo para compilar y entrenar.
"""

inputs = Input(shape=input_shape) # Shape: (batch_size, sequence_length, num_fe
x = inputs

# Determinar la dimensión de embedding:
# Si embedding_dim no se especifica, usa el `head_size` para que sea consistente
if embedding_dim is None:
    embedding_dim = head_size

x = PositionalEmbedding(output_dim=embedding_dim, seq_length=seq_length)(x) # S

# Apilar bloques de codificador Transformer
for _ in range(num_transformer_blocks):
    x = transformer_block(x, head_size, num_heads, ff_dim, dropout) # Shape: (b

x = GlobalAveragePooling1D()(x) # Shape: (batch_size, sequence_length, embeddin

# Capa MLP
for dim in mlp_units:
    x = Dense(dim, activation="relu")(x) # Shape: (batch_size, embedding_dim) ->
    x = Dropout(mlp_dropout)(x) # Shape: (batch_size, dim) -> (batch_size, dim)

outputs = Dense(1)(x) # Shape: (batch_size, dim) -> (batch_size, 1)

return Model(inputs, outputs)

```

Creamos el "nuevo" modelo:

```
In [27]: input_shape = X_train.shape[1:]
model = build_model_with_embeddings(
    input_shape,
    head_size=256,
    num_heads=4,
    ff_dim=4,
    num_transformer_blocks=4,
    mlp_units=[128],
    seq_length=WINDOW_SIZE,
    dropout=0.25,
    mlp_dropout=0.4,
    embedding_dim=EMBEDDING_DIM
)
```

WARNING:tensorflow:From e:\Documentos\Git Repositories\uba-mia-ast2\.venv\Lib\site-packages\keras\src\backend\tensorflow\core.py:232: The name tf.placeholder is deprecated. Please use tf.compat.v1.placeholder instead.

Mostramos el resumen:

```
In [28]: model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 5, 1)	0	-
positional_embedding... (PositionalEmbedding)	(None, 5, 8)	56	input_layer_1[0]...
layer_normalization... (LayerNormalization)	(None, 5, 8)	16	positional_embed...
multi_head_attention... (MultiHeadAttention)	(None, 5, 8)	35,848	layer_normalizat... layer_normalizat...
dropout_14 (Dropout)	(None, 5, 8)	0	multi_head_atten...
add_8 (Add)	(None, 5, 8)	0	dropout_14[0][0], positional_embed...
layer_normalization... (LayerNormalization)	(None, 5, 8)	16	add_8[0][0]
conv1d_8 (Conv1D)	(None, 5, 4)	36	layer_normalizat...

Finalmente, compilamos el modelo y lo entrenamos:

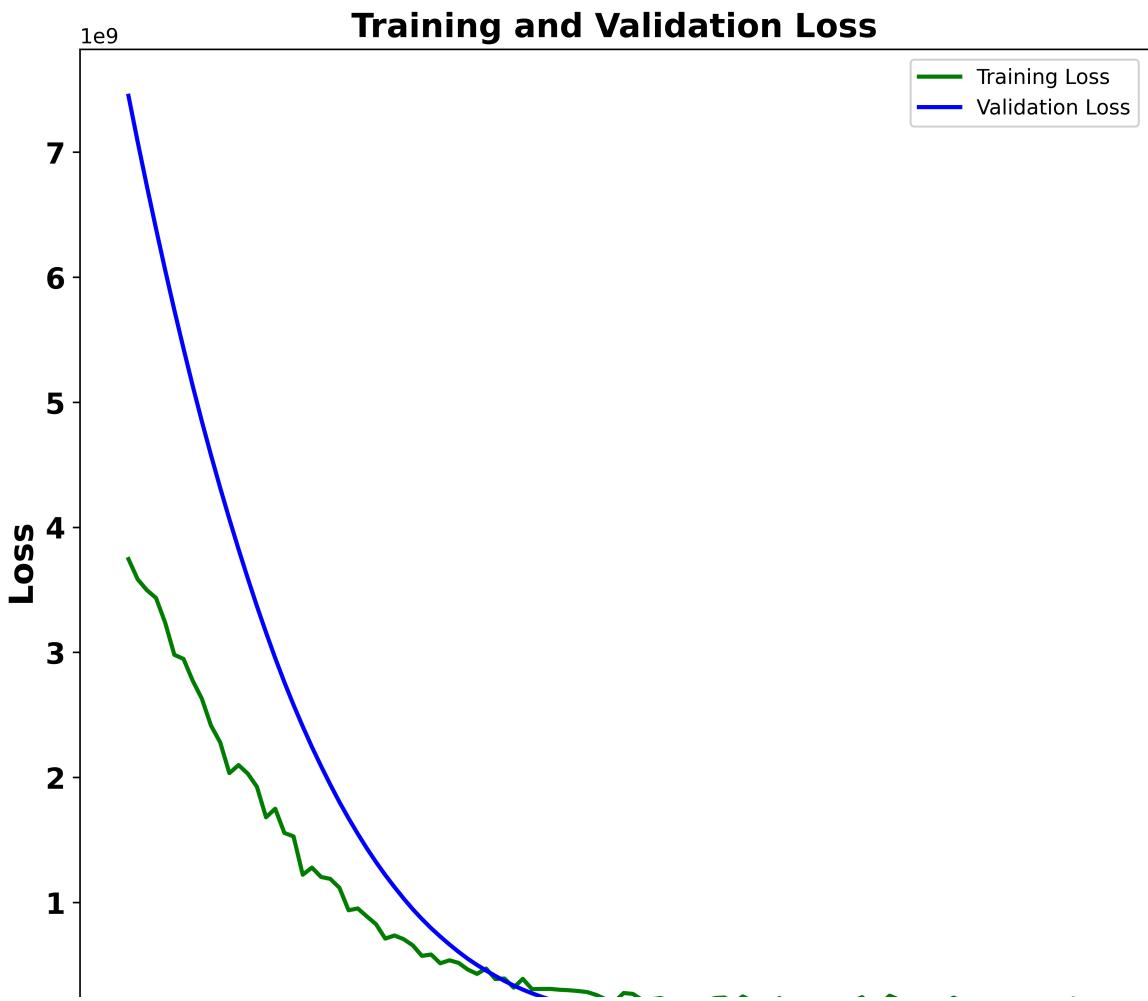
```
In [29]: model.compile(
    loss=MeanSquaredError(), optimizer=Adam(learning_rate=1e-4), metrics=[MeanAbsol
)

history = model.fit(X_train, y_train, validation_split=0.2, epochs=500, batch_size=
```

```
Epoch 1/500
5/5 6s 118ms/step - loss: 3745746176.0000 - mean_absolute_error: 59417.3438 - val_loss: 7450994176.0000 - val_mean_absolute_error: 86252.1484
Epoch 2/500
5/5 0s 34ms/step - loss: 3584442624.0000 - mean_absolute_error: 58189.7344 - val_loss: 7083529216.0000 - val_mean_absolute_error: 84096.5781
Epoch 3/500
5/5 0s 32ms/step - loss: 3497890816.0000 - mean_absolute_error: 57348.5156 - val_loss: 6730481152.0000 - val_mean_absolute_error: 81972.1953
Epoch 4/500
5/5 0s 33ms/step - loss: 3434803968.0000 - mean_absolute_error: 56559.7656 - val_loss: 6389161984.0000 - val_mean_absolute_error: 79864.6641
Epoch 5/500
5/5 0s 32ms/step - loss: 3237461248.0000 - mean_absolute_error: 54969.1289 - val_loss: 6057072640.0000 - val_mean_absolute_error: 77759.3047
Epoch 6/500
5/5 0s 33ms/step - loss: 2980279040.0000 - mean_absolute_error: 53075.0625 - val_loss: 5737446912.0000 - val_mean_absolute_error: 75677.6484
Epoch 7/500
5/5 0s 33ms/step - loss: 2947464448.0000 - mean_absolute_error: 52717.8008 - val_loss: 5429929984.0000 - val_mean_absolute_error: 73619.3047
Epoch 8/500
5/5 0s 33ms/step - loss: 2773769216.0000 - mean_absolute_error: 50736.4922 - val_loss: 5131864576.0000 - val_mean_absolute_error: 71567.7188
Epoch 9/500
5/5 0s 34ms/step - loss: 2629158400.0000 - mean_absolute_error: 49513.3906 - val_loss: 4847659008.0000 - val_mean_absolute_error: 69555.1797
Epoch 10/500
5/5 0s 33ms/step - loss: 2413897472.0000 - mean_absolute_error:
```

Mostramos el progreso del entrenamiento:

```
In [30]: plot_loss(history, "resources/embedding_model_loss.jpeg")
```



Evaluamos el modelo en el conjunto de test:

```
In [31]: test_loss = model.evaluate(X_test, y_test, verbose=1, return_dict=True)
pprint(test_loss)
```

2/2 ━━━━━━ 0s 20ms/step - loss: 13550875.0000 - mean_absolute_error: 2
856.5977
2/2 ━━━━━━ 0s 20ms/step - loss: 13550875.0000 - mean_absolute_error: 2
856.5977
{'loss': 13550875.0, 'mean_absolute_error': 2856.59765625}

Calculamos las métricas de evaluación:

```
In [32]: pred_test = model.predict(X_test)
metrics_dict_test = calculate_metrics(y_test, pred_test)
print("\nTest Set Metrics:")
for metric_name, metric_value in metrics_dict_test.items():
    print(f"Score ({metric_name}): {metric_value}")
```

2/2 ————— 1s 257ms/step

Test Set Metrics:

```
Score (RMSE): 3681.151605436321
Score (MAPE): 3.3357610550518024
Score (MAE): 2856.598098214285
Score (MSE): 13550877.142206404
Score (sMAPE): 3.3292887925528225
Score (RRMSE): 4.323217428686552
Score (Quantile Loss): 1428.2990491071425
```

Guardamos las métricas de evaluación en el diccionario:

```
In [33]: df_line = pd.DataFrame({
    "model": "CustTransfEmb",
    "mae": metrics_dict_test["MAE"],
    "mape": metrics_dict_test["MAPE"],
    "rmse": metrics_dict_test["RMSE"]
}, index=[0])

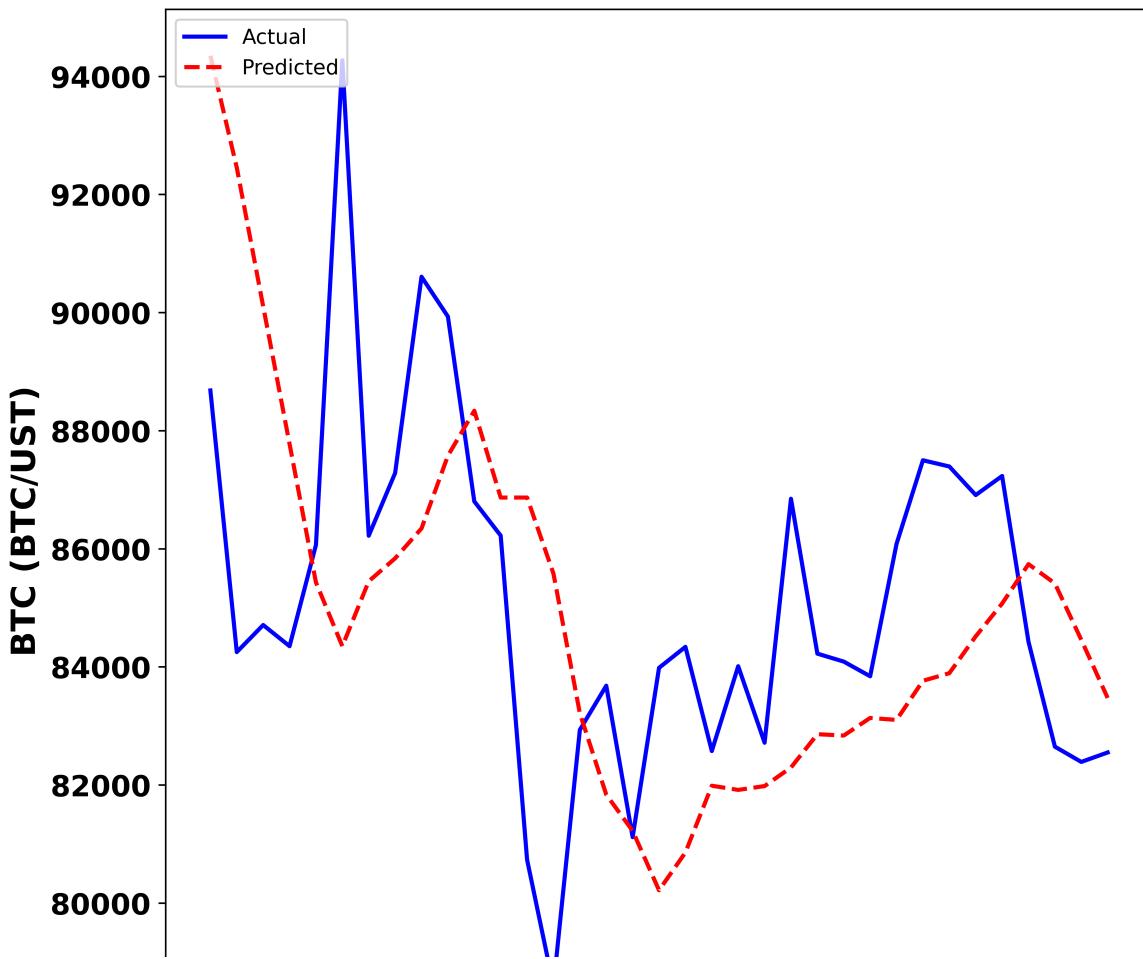
df_metrics = pd.concat([df_metrics, df_line], ignore_index=True)
df_metrics
```

	model	mae	mape	rmse
0	Naive	1735.726500	1.888647	2434.504211
1	ARIMA(1,1,35)	1891.662281	2.063588	2546.257391
2	ARIMA(0,1,0)	1735.726500	1.894494	2434.504211
3	Prophet	2582.551322	2.871448	3621.703076
4	XGBoost	4319.031819	4.609317	5555.304516
5	LSTM	2432.965561	2.645095	3192.781096
6	CustTransf	2437.519884	2.837959	3324.778151
7	CustTransfEmb	2856.598098	3.335761	3681.151605

Realizamos la predicción y graficamos los resultados:

```
In [34]: plot_actual_vs_predicted(y_test, pred_test, "resources/embedding_model_predict.jpeg")
```

Actual vs Predicted



4. Estado del arte

4.1. Caso de uso: TimesFM

En esta sección, se presenta el análisis de la serie que se viene trabajando pero con el modelo *TimesFM*. Se utiliza `TimesFmModelForPrediction` de *HuggingFace*.

`TimesFmModelForPrediction` espera:

- → `past_values` : Una lista de tensores, donde cada tensor representa una serie temporal individual. *El modelo está diseñado para pronosticar múltiples series temporales a la vez, por lo que past_values es una lista*. Cada tensor en la lista debe tener la forma `(sequence_length,)`.
- → `freq` : Una lista (o un tensor) de índices de frecuencia, uno para cada serie temporal en `past_values`. `TimesFM` utiliza estas frecuencias para adaptar su codificación posicional y otros aspectos del modelo a la periodicidad de los datos. Los valores típicos son:
 - 0: Alta frecuencia (ej. diario, horario).
 - 1: Frecuencia media (ej. semanal, mensual).

- 2: Baja frecuencia (ej. trimestral, anual).

En el ejemplo de *HuggingFace*, se utilizan estos datos como prueba:

```
# Create dummy inputs
forecast_input = [
    np.sin(np.linspace(0, 20, 100)),
    np.sin(np.linspace(0, 20, 200)),
    np.sin(np.linspace(0, 20, 400)),
]
frequency_input = [0, 1, 2]

# Convert inputs to sequence of tensors
forecast_input_tensor = [
    torch.tensor(ts, dtype=torch.bfloat16).to("cuda" if
torch.cuda.is_available() else "cpu") for ts in forecast_input
]
frequency_input_tensor = torch.tensor(frequency_input,
dtype=torch.long).to(
    "cuda" if torch.cuda.is_available() else "cpu"
)
```

En donde podemos observar que `forecast_input` es una lista de series temporales, y `frequency_input` es una lista de frecuencias asociadas a cada serie temporal. Para este caso, el conjunto `spots_train` y `spots_test` ya tienen la forma adecuada (serie univariada), por lo que simplemente debemos crear la lista de tensores y las frecuencias asociadas. A diferencia de los casos anteriores, en donde se formateaba el conjunto para una ventana de tiempo.

Convertimos los datos a tensores:

```
In [35]: forecast_input = np.array(spots_train).flatten()
```

Creamos los tensores y lo tiramos a la GPU si está disponible:

```
In [36]: forecast_input_tensor = [torch.tensor(forecast_input, dtype=torch.bfloat16).to(DEVICE)
frequency_input_tensor = torch.tensor([0], dtype=torch.long).to(DEVICE)
print(f"Input tensor shape: {forecast_input_tensor[0].shape}")
print(f"Frequency tensor shape: {frequency_input_tensor.shape}")
```

```
Input tensor shape: torch.Size([356])
Frequency tensor shape: torch.Size([1])
```

Cargamos el modelo:

```
In [37]: model = TimesFmModelForPrediction.from_pretrained(
    "google/timesfm-2.0-500m-pytorch",
    torch_dtype=torch.bfloat16,
    attn_implementation="sdpd",
    device_map=DEVICE,
)
```

```
config = model.config
```

Imprimimos el resumen del modelo:

```
In [38]: print(model)
```

```
TimesFmModelForPrediction(  
    (decoder): TimesFmModel(  
        (input_ff_layer): TimesFmResidualBlock(  
            (input_layer): Linear(in_features=64, out_features=1280, bias=True)  
            (activation): SiLU()  
            (output_layer): Linear(in_features=1280, out_features=1280, bias=True)  
            (residual_layer): Linear(in_features=64, out_features=1280, bias=True)  
        )  
        (freq_emb): Embedding(3, 1280)  
        (layers): ModuleList(  
            (0-49): 50 x TimesFmDecoderLayer(  
                (self_attn): TimesFmAttention(  
                    (q_proj): Linear(in_features=1280, out_features=1280, bias=True)  
                    (k_proj): Linear(in_features=1280, out_features=1280, bias=True)  
                    (v_proj): Linear(in_features=1280, out_features=1280, bias=True)  
                    (o_proj): Linear(in_features=1280, out_features=1280, bias=True)  
                )  
                (mlp): TimesFmMLP(  
                    (gate_proj): Linear(in_features=1280, out_features=1280, bias=True)  
                    (down_proj): Linear(in_features=1280, out_features=1280, bias=True)  
                    (layer_norm): LayerNorm((1280,), eps=1e-06, elementwise_affine=True)  
                )  
                (input_layernorm): TimesFmRMSNorm((1280,), eps=1e-06)  
            )  
        )  
        (horizon_ff_layer): TimesFmResidualBlock(  
            (input_layer): Linear(in_features=1280, out_features=1280, bias=True)  
            (activation): SiLU()  
    )
```

Mostramos la configuración del modelo:

```
In [39]: print(config)
```

```
TimesFmConfig {  
    "architectures": [  
        "TimesFmModelForPrediction"  
    ],  
    "attention_dropout": 0.0,  
    "context_length": 2048,  
    "freq_size": 3,  
    "head_dim": 80,  
    "hidden_size": 1280,  
    "horizon_length": 128,  
    "initializer_range": 0.02,  
    "intermediate_size": 1280,  
    "max_timescale": 10000,  
    "min_timescale": 1,  
    "model_type": "timesfm",  
    "num_attention_heads": 16,  
    "num_hidden_layers": 50,  
    "pad_val": 1123581321.0,  
    "patch_length": 32,  
    "quantiles": [  
        0.1,  
        0.2,  
        0.3,  
        0.4,  
        0.5,  
        0.6,  
        0.7,  
        0.8,  
        0.9  
    ]  
}
```

 **Idea:** Observamos que el modelo tiene un horizonte de predicción de 128 pasos, por lo que está contemplado nuestro conjunto de test, que tiene cerca de 40 pasos (días):

```
In [40]: assert config.horizon_length > len(spots_test)
```

Realizamos la predicción:

```
In [41]: # Get predictions from the pre-trained model  
with torch.no_grad():  
    outputs = model(past_values=forecast_input_tensor, freq=frequency_input_tensor,  
    point_forecast_conv = outputs.mean_predictions.float().cpu().numpy()  
    quantile_forecast_conv = outputs.full_predictions.float().cpu().numpy()  
  
    # Print the shape of the predictions  
print("Point Forecast Shape:", point_forecast_conv.shape)  
print("Quantile Forecast Shape:", quantile_forecast_conv.shape)
```

```
Point Forecast Shape: (1, 128)  
Quantile Forecast Shape: (1, 128, 10)
```

Finalmente, truncamos la predicción a los primeros 40 pasos y procesamos los resultados:

```
In [42]: y_true = np.array(spots_test).flatten()
pred_test = point_forecast_conv.flatten()[:len(y_true)]
```

Calculamos las métricas de evaluación:

```
In [43]: metrics_dict_test = calculate_metrics(y_true, pred_test)
print("\nTest Set Metrics:")
for metric_name, metric_value in metrics_dict_test.items():
    print(f"Score ({metric_name}): {metric_value}")
```

Test Set Metrics:
Score (RMSE): 3870.8257111206262
Score (MAPE): 3.819422342604954
Score (MAE): 3279.7782500000003
Score (MSE): 14983291.6858725
Score (sMAPE): 3.777163122355528
Score (RRMSE): 4.476180061450597
Score (Quantile Loss): 1639.8891250000001

Guardamos las métricas de evaluación en el diccionario:

```
In [44]: df_line = pd.DataFrame({
    "model": "TimesFM",
    "mae": metrics_dict_test["MAE"],
    "mape": metrics_dict_test["MAPE"],
    "rmse": metrics_dict_test["RMSE"]
}, index=[0])

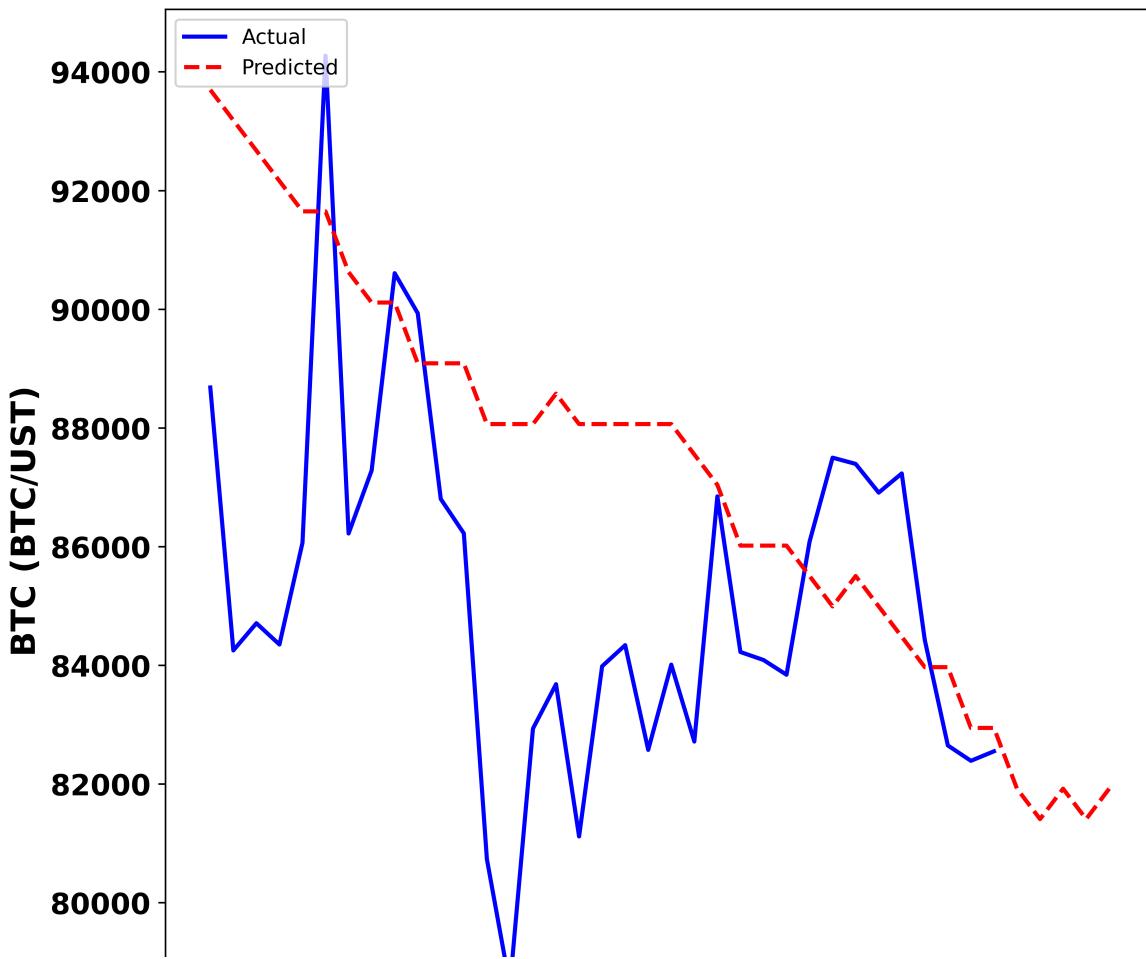
df_metrics = pd.concat([df_metrics, df_line], ignore_index=True)
df_metrics
```

	model	mae	mape	rmse
0	Naive	1735.726500	1.888647	2434.504211
1	ARIMA(1,1,35)	1891.662281	2.063588	2546.257391
2	ARIMA(0,1,0)	1735.726500	1.894494	2434.504211
3	Prophet	2582.551322	2.871448	3621.703076
4	XGBoost	4319.031819	4.609317	5555.304516
5	LSTM	2432.965561	2.645095	3192.781096
6	CustTransf	2437.519884	2.837959	3324.778151
7	CustTransfEmb	2856.598098	3.335761	3681.151605
8	TimesFM	3279.778250	3.819422	3870.825711

Graficamos los resultados:

```
In [45]: plot_actual_vs_predicted(y_test, pred_test, "resources/timesfm_model_predict.jpeg")
```

Actual vs Predicted



4.2. Caso de uso: PatchTST

In [46]:

```
# --- Definir la clase Dataset para series temporales ---
class TimeSeriesDataset(Dataset):
    def __init__(self, data, context_length, forecast_horizon):
        self.data = data
        self.context_length = context_length
        self.forecast_horizon = forecast_horizon

    def __len__(self):
        return max(0, len(self.data) - self.context_length - self.forecast_horizon)

    def __getitem__(self, idx):
        x = self.data[idx : idx + self.context_length]
        y = self.data[idx + self.context_length : idx + self.context_length + self.forecast_horizon]
        return torch.tensor(x, dtype=torch.float32), torch.tensor(y, dtype=torch.float32)

# --- Parámetros ---
context_length = 24
forecast_horizon = 1
batch_size = 16

# --- Normalizar la serie ---
scaler = StandardScaler()
df['valor_normalizado'] = scaler.fit_transform(df[['Close']])
```

```
# --- Extraer la serie normalizada ---
series = df['valor_normalizado'].values
```

```
In [47]: # --- Dividir en train y test ---
N = len(series)
train_size = int(N * 0.9)

train_series = series[:train_size]
test_series = series[train_size:]

print(f"Tamaño total: {N}, Train: {len(train_series)}, Test: {len(test_series)}")

# --- Crear datasets ---
train_dataset = TimeSeriesDataset(train_series, context_length, forecast_horizon)
test_dataset = TimeSeriesDataset(test_series, context_length, forecast_horizon)

print(f"Ejemplos train: {len(train_dataset)}, Ejemplos test: {len(test_dataset)}")

# --- Crear DataLoaders ---
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Ahora train_loader y test_loader están listos para usar en entrenamiento y evaluación
```

Tamaño total: 396, Train: 356, Test: 40

Ejemplos train: 331, Ejemplos test: 15

```
In [48]: # Crear la configuración del modelo
config = PatchTSTConfig(
    context_length=24,
    prediction_length=12,
    d_model=64,
    num_input_channels=1,           # Univariada
    num_targets=1,
    patch_length=12,
    patch_stride=6,
    num_hidden_layers=1,
    num_attention_heads=4,
    dropout=0.1
)

# Instanciar el modelo
model = PatchTSTForPrediction(config)
```

```
In [49]: # --- Parámetros ---
num_epochs = 50
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# --- Mover modelo a dispositivo ---
model = model.to(device)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

train_losses = []
test_losses = []
```

```
for epoch in range(num_epochs):
    model.train()
    train_loss = 0.0

    for x_batch, y_batch in train_loader:
        x_batch = x_batch.to(device).unsqueeze(-1)
        y_batch = y_batch.to(device)

        optimizer.zero_grad()
        output = model(past_values=x_batch)
        preds = output.prediction_outputs.squeeze(-1)
        loss = criterion(preds, y_batch)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * x_batch.size(0)

    train_loss /= len(train_loader.dataset)
    train_losses.append(train_loss)

model.eval()
test_loss = 0.0
with torch.no_grad():
    for x_batch, y_batch in test_loader:
        x_batch = x_batch.to(device).unsqueeze(-1)
        y_batch = y_batch.to(device)

        output = model(past_values=x_batch)
        preds = output.prediction_outputs.squeeze(-1)
        loss = criterion(preds, y_batch)

        test_loss += loss.item() * x_batch.size(0)

    test_loss /= len(test_loader.dataset)
    test_losses.append(test_loss)

print(f"Epoch {epoch+1}/{num_epochs} | Train Loss: {train_loss:.4f} | Test Loss: {test_loss:.4f}")

# Graficar la evolución del error
plt.figure(figsize=(8,5))
plt.plot(range(1, num_epochs+1), train_losses, label='Train Loss')
plt.plot(range(1, num_epochs+1), test_losses, label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss (MSE)')
plt.title('Evolución del error durante el entrenamiento')
plt.legend()
plt.grid(True)
plt.show()
```

```
e:\Documentos\Git Repositories\uba-mia-ast2\.venv\Lib\site-packages\torch\nn\modules\loss.py:610: UserWarning: Using a target size (torch.Size([16, 1])) that is different to the input size (torch.Size([16, 12])). This will likely lead to incorrect results due to broadcasting. Please ensure they have the same size.  
    return F.mse_loss(input, target, reduction=self.reduction)  
e:\Documentos\Git Repositories\uba-mia-ast2\.venv\Lib\site-packages\torch\nn\modules\loss.py:610: UserWarning: Using a target size (torch.Size([11, 1])) that is different to the input size (torch.Size([11, 12])). This will likely lead to incorrect results due to broadcasting. Please ensure they have the same size.  
    return F.mse_loss(input, target, reduction=self.reduction)  
e:\Documentos\Git Repositories\uba-mia-ast2\.venv\Lib\site-packages\torch\nn\modules\loss.py:610: UserWarning: Using a target size (torch.Size([15, 1])) that is different to the input size (torch.Size([15, 12])). This will likely lead to incorrect results due to broadcasting. Please ensure they have the same size.  
    return F.mse_loss(input, target, reduction=self.reduction)  
Epoch 1/50 | Train Loss: 0.1131 | Test Loss: 0.0242  
Epoch 2/50 | Train Loss: 0.1048 | Test Loss: 0.0227  
Epoch 3/50 | Train Loss: 0.0805 | Test Loss: 0.0185  
Epoch 4/50 | Train Loss: 0.0486 | Test Loss: 0.0300  
Epoch 5/50 | Train Loss: 0.0310 | Test Loss: 0.0267  
Epoch 6/50 | Train Loss: 0.0285 | Test Loss: 0.0333  
Epoch 7/50 | Train Loss: 0.0283 | Test Loss: 0.0294  
Epoch 8/50 | Train Loss: 0.0249 | Test Loss: 0.0307  
Epoch 9/50 | Train Loss: 0.0250 | Test Loss: 0.0299  
Epoch 10/50 | Train Loss: 0.0241 | Test Loss: 0.0314  
Epoch 11/50 | Train Loss: 0.0258 | Test Loss: 0.0306  
Epoch 12/50 | Train Loss: 0.0235 | Test Loss: 0.0292  
Epoch 13/50 | Train Loss: 0.0275 | Test Loss: 0.0267  
Epoch 14/50 | Train Loss: 0.0255 | Test Loss: 0.0270
```

```
In [50]: torch.save(model, "patchtst_model_complete.pth")
```

```
In [51]: model = torch.load("patchtst_model_complete.pth", weights_only=False)  
model.to(device)  
model.eval()
```

```
Out[51]: PatchTSTForPrediction(
    (model): PatchTSTModel(
        (scaler): PatchTSTScaler(
            (scaler): PatchTSTStdScaler()
        )
        (patchifier): PatchTSTPatchify()
        (masking): Identity()
        (encoder): PatchTSTEEncoder(
            (embedder): PatchTSTEEmbedding(
                (input_embedding): Linear(in_features=12, out_features=64, bias=True)
            )
            (positional_encoder): PatchTSTPositionalEncoding(
                (positional_dropout): Identity()
            )
        )
        (layers): ModuleList(
            (0): PatchTSTEEncoderLayer(
                (self_attn): PatchTSTAttention(
                    (k_proj): Linear(in_features=64, out_features=64, bias=True)
                    (v_proj): Linear(in_features=64, out_features=64, bias=True)
                    (q_proj): Linear(in_features=64, out_features=64, bias=True)
                    (out_proj): Linear(in_features=64, out_features=64, bias=True)
                )
                (dropout_path1): Identity()
                (norm_sublayer1): PatchTSTBatchNorm(
                    (batchnorm): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
                )
                (ff): Sequential(
                    (0): Linear(in_features=64, out_features=512, bias=True)
                )
            )
        )
    )
)
```

```
In [52]: df_TST = pd.read_csv('resources/BTCUSDT_1D.csv')
df_TST.head()
```

```
Out[52]:   Open      Time     Close
0  2024-03-01  62387.90
1  2024-03-02  61987.28
2  2024-03-03  63113.97
3  2024-03-04  68245.71
4  2024-03-05  63724.01
```

```
In [53]: # Parámetros
input_length = 24
forecast_steps = 40
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 1. Extraer valores de la serie
valores = df_TST.iloc[:, 1].values.astype(np.float32)

# 2. Normalizar la serie completa con el scaler ya entrenado
valores_norm = scaler.transform(valores.reshape(-1, 1)).flatten()
```

```

# 3. Crear ventanas deslizantes para los últimos forecast_steps puntos
serie_len = len(valores)
forecast_steps = min(forecast_steps, serie_len - input_length) # asegurar no excede

start_idx = serie_len - forecast_steps - input_length
end_idx = serie_len - input_length

X = []
y_real = []

for i in range(start_idx, end_idx):
    x_i = valores_norm[i : i + input_length]
    y_i = valores[i + input_length]
    X.append(x_i)
    y_real.append(y_i)

X = np.array(X)
y_real = np.array(y_real)

# 4. Preparar tensores y predecir
X_tensor = torch.tensor(X, dtype=torch.float32).unsqueeze(-1).to(device)

model.eval()
with torch.no_grad():
    output = model(X_tensor)
    y_pred_norm = output.prediction_outputs[:, -1, 0].cpu().numpy() # último paso

# 5. Desnormalizar predicciones
y_pred = scaler.inverse_transform(y_pred_norm.reshape(-1, 1)).flatten()

# 6. Calcular métricas
rmse_TST = root_mean_squared_error(y_real, y_pred)
mae_TST = mean_absolute_error(y_real, y_pred)
mape_TST = np.mean(np.abs((y_real - y_pred) / y_real)) * 100

metrics_dict_test = calculate_metrics(y_real, y_pred)
print("\nTest Set Metrics:")
for metric_name, metric_value in metrics_dict_test.items():
    print(f"Score ({metric_name}): {metric_value}")

```

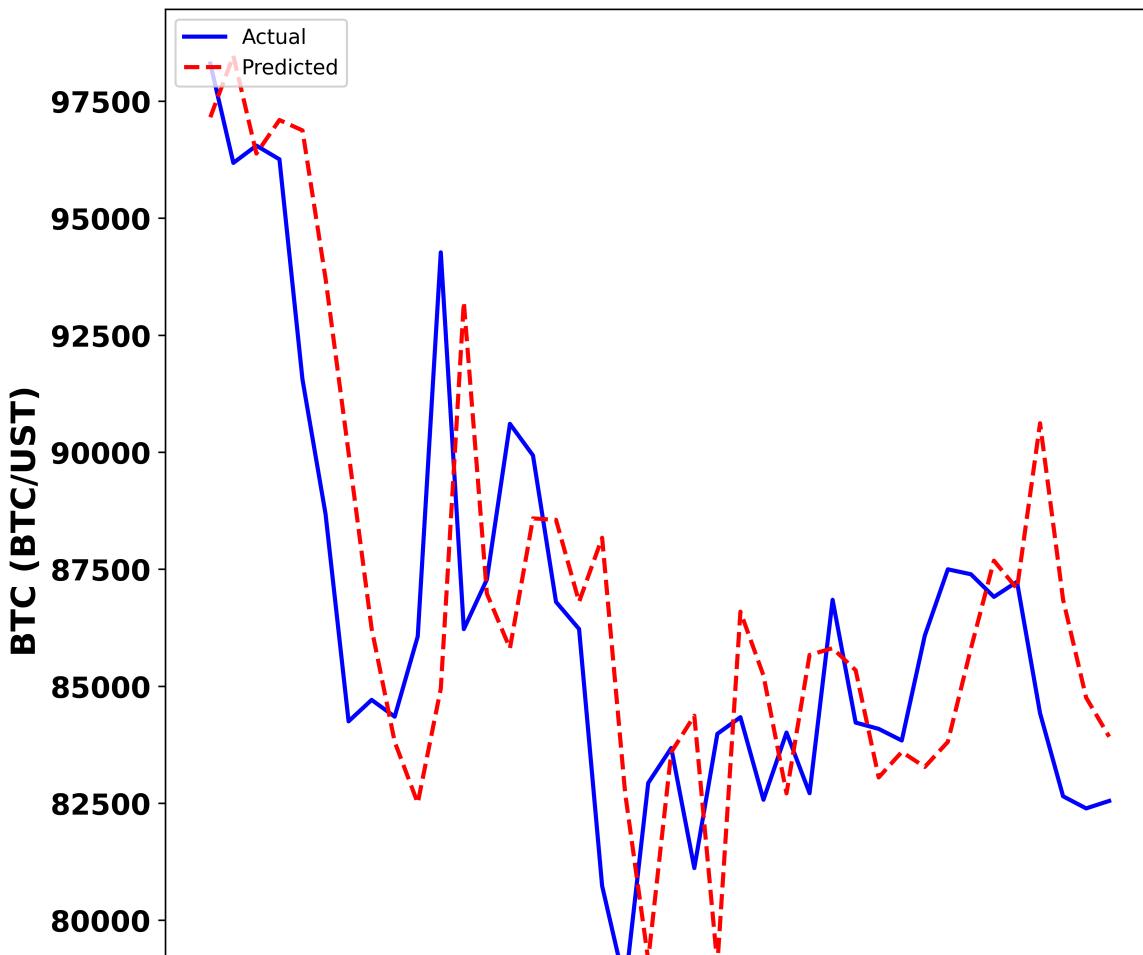
Test Set Metrics:

Score (RMSE): 3566.8814109807463
 Score (MAPE): 3.216271847486496
 Score (MAE): 2764.9443359375
 Score (MSE): 12722643.0
 Score (sMAPE): 3.193509101867676
 Score (RRMSE): 4.124702280485173
 Score (Quantile Loss): 1382.47216796875

e:\Documentos\Git Repositories\uba-mia-ast2\.venv\Lib\site-packages\sklearn\utils\validation.py:2739: UserWarning: X does not have valid feature names, but StandardScaler was fitted with feature names
 warnings.warn(

In [54]: plot_actual_vs_predicted(y_real, y_pred, "resources/patchtst_model_predict.jpeg")

Actual vs Predicted



```
In [55]: metrics_dict_test_TST = {
    "MAE": mae_TST,
    "MAPE": mape_TST,
    "RMSE": rmse_TST
}

df_line_TST = pd.DataFrame({
    "model": "PatchTST",
    "mae": metrics_dict_test_TST["MAE"],
    "mape": metrics_dict_test_TST["MAPE"],
    "rmse": metrics_dict_test_TST["RMSE"]
}, index=[0])

df_metrics = pd.concat([df_metrics, df_line_TST], ignore_index=True)
df_metrics
```

Out[55]:

	model	mae	mape	rmse
0	Naive	1735.726500	1.888647	2434.504211
1	ARIMA(1,1,35)	1891.662281	2.063588	2546.257391
2	ARIMA(0,1,0)	1735.726500	1.894494	2434.504211
3	Prophet	2582.551322	2.871448	3621.703076
4	XGBoost	4319.031819	4.609317	5555.304516
5	LSTM	2432.965561	2.645095	3192.781096
6	CustTransf	2437.519884	2.837959	3324.778151
7	CustTransfEmb	2856.598098	3.335761	3681.151605
8	TimesFM	3279.778250	3.819422	3870.825711
9	PatchTST	2764.944336	3.216272	3566.881348

5. Resultados

In [56]:

```
def plot_metrics_comparison_bar(df: pd.DataFrame, filename: str, fig_size = (10, 6)
    """
    Grafica una comparación de métricas (MAE) entre diferentes modelos,
    ordenándolos de menor a mayor MAE, y guarda la figura en un archivo.

    Args:
        df (pd.DataFrame): DataFrame que contiene las métricas de los modelos.
        filename (str): Nombre del archivo donde se guardará la figura.
        fig_size (tuple): Tupla para el tamaño de la figura (ancho, alto).

    Returns:
        None
    """
    df_grouped = df.groupby('model')['mae'].mean().reset_index()

    df_sorted = df_grouped.sort_values(by='mae', ascending=True)

    models = df_sorted['model'].tolist()
    mae_values = df_sorted['mae'].tolist()

    x_pos = np.arange(len(models))

    plt.figure(figsize=fig_size, dpi=600)
    plt.bar(x_pos, mae_values, color='skyblue')

    plt.xlabel('Modelo')
    plt.ylabel('Error Absoluto Medio (MAE)')
    plt.title('Comparación de Métrica MAE por Modelo')
    plt.xticks(x_pos, models, rotation=45, ha='right') # Rotar las etiquetas para q
    # Añadir los valores de MAE encima de cada barra
```

```

for i, v in enumerate(mae_values):
    plt.text(x_pos[i], v + 50, f"{v:.2f}", ha='center', va='bottom', fontsize=9)

plt.grid(axis='y', linestyle='--', alpha=0.3) # Añade una cuadrícula suave en e
plt.tight_layout() # Ajusta el diseño para que no se corten las etiquetas
plt.savefig(filename, format='jpeg', dpi=600)
plt.show()

def plot_metrics_radial(df: pd.DataFrame, filename: str, fig_size=(8, 8)) -> None:
    """
    Grafica una comparación de múltiples métricas (MAE, MAPE, RMSE) para
    diferentes modelos en un gráfico radial (o de araña/radar) y guarda la figura.
    Las métricas se normalizan para una mejor comparación.

    Args:
        df (pd.DataFrame): DataFrame que contiene las métricas de los modelos.
        filename (str): Nombre del archivo donde se guardará la figura.
        fig_size (tuple): Tupla para el tamaño de la figura (ancho, alto).

    Returns:
        None
    """
    metrics = ['mae', 'mape', 'rmse']

    # Manejo de duplicados
    df_processed = df.groupby('model')[metrics].mean().reset_index()

    # Normalizar las métricas para que estén entre 0 y 1
    df_normalized = df_processed.copy()
    for metric in metrics:
        min_val = df_normalized[metric].min()
        max_val = df_normalized[metric].max()
        # Evitar división por cero si todos los valores son iguales
        if max_val == min_val:
            df_normalized[metric] = 0.5 # Valor neutro si no hay variación
        else:
            # Invertir la escala para que "mejor" (menor error) esté más cerca del
            df_normalized[metric] = 1 - ((df_normalized[metric] - min_val) / (max_val - min_val))

    models = df_normalized['model'].tolist()

    num_metrics = len(metrics)

    # Calcular el ángulo para cada métrica
    angles = np.linspace(0, 2 * np.pi, num_metrics, endpoint=False).tolist()

    # Duplicar el primer ángulo y la primera métrica para cerrar el círculo del grá
    angles += angles[:1]

    # Configurar el gráfico
    fig, ax = plt.subplots(figsize=fig_size, subplot_kw=dict(polar=True), dpi=600)

    # Colores para cada modelo
    colors = plt.colormaps.get_cmap('tab10')

    # Graficar cada modelo

```

```

for i, model in enumerate(models):
    values = df_normalized[df_normalized['model'] == model][metrics].values[0].
    values += values[:1] # Cerrar el círculo para este modelo
    ax.plot(angles, values, linewidth=2, linestyle='solid', label=model, color=
    ax.fill(angles, values, color=colors(i), alpha=0.25) # Rellenar el área

# Configurar las etiquetas de las métricas
ax.set_theta_offset(np.pi / 2) # Rotar el inicio a la parte superior
ax.set_theta_direction(-1)      # Dirección en sentido horario
ax.set_xticks(angles[:-1])
ax.set_xticklabels(metrics)

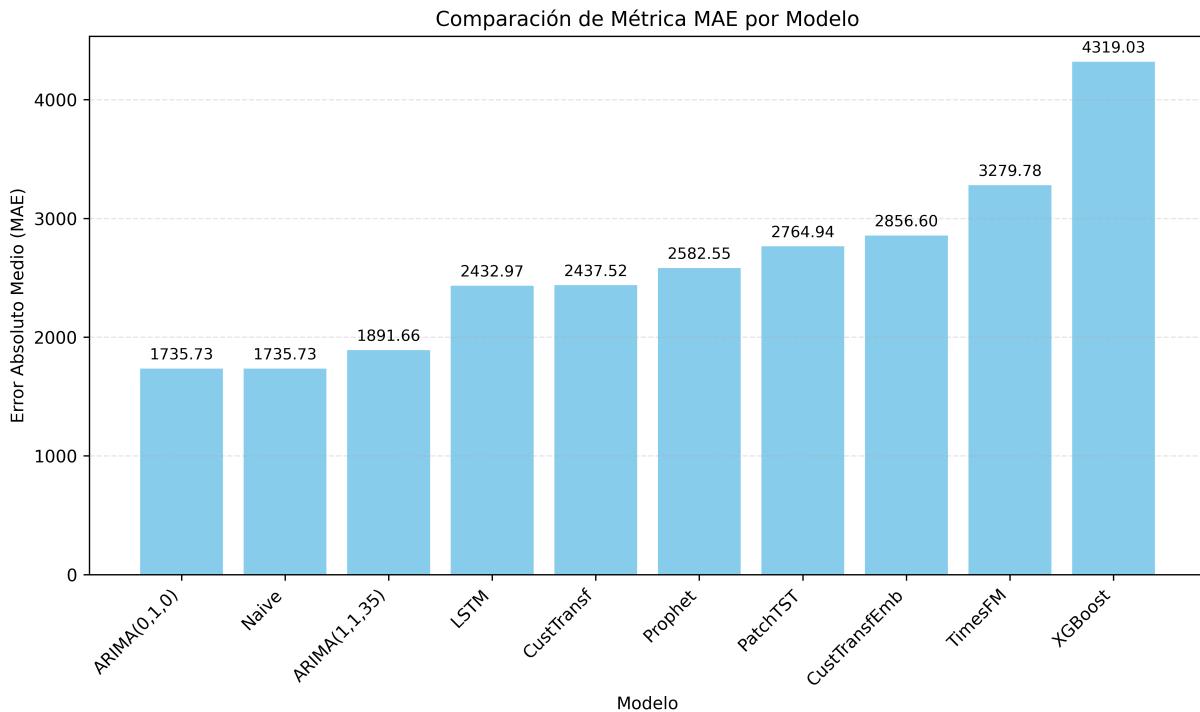
# Configurar el eje radial
ax.set_yticks([0.2, 0.4, 0.6, 0.8, 1.0])
ax.set_yticklabels(['80%', '60%', '40%', '20%', '0% (Mejor)'], color='gray', si
ax.set_ylim(0, 1) # Rango del eje radial

# Título y leyenda
ax.set_title('Comparación de Modelos por Métricas', va='bottom', y=1.1)
ax.legend(loc='upper right', bbox_to_anchor=(1.3, 1.1))

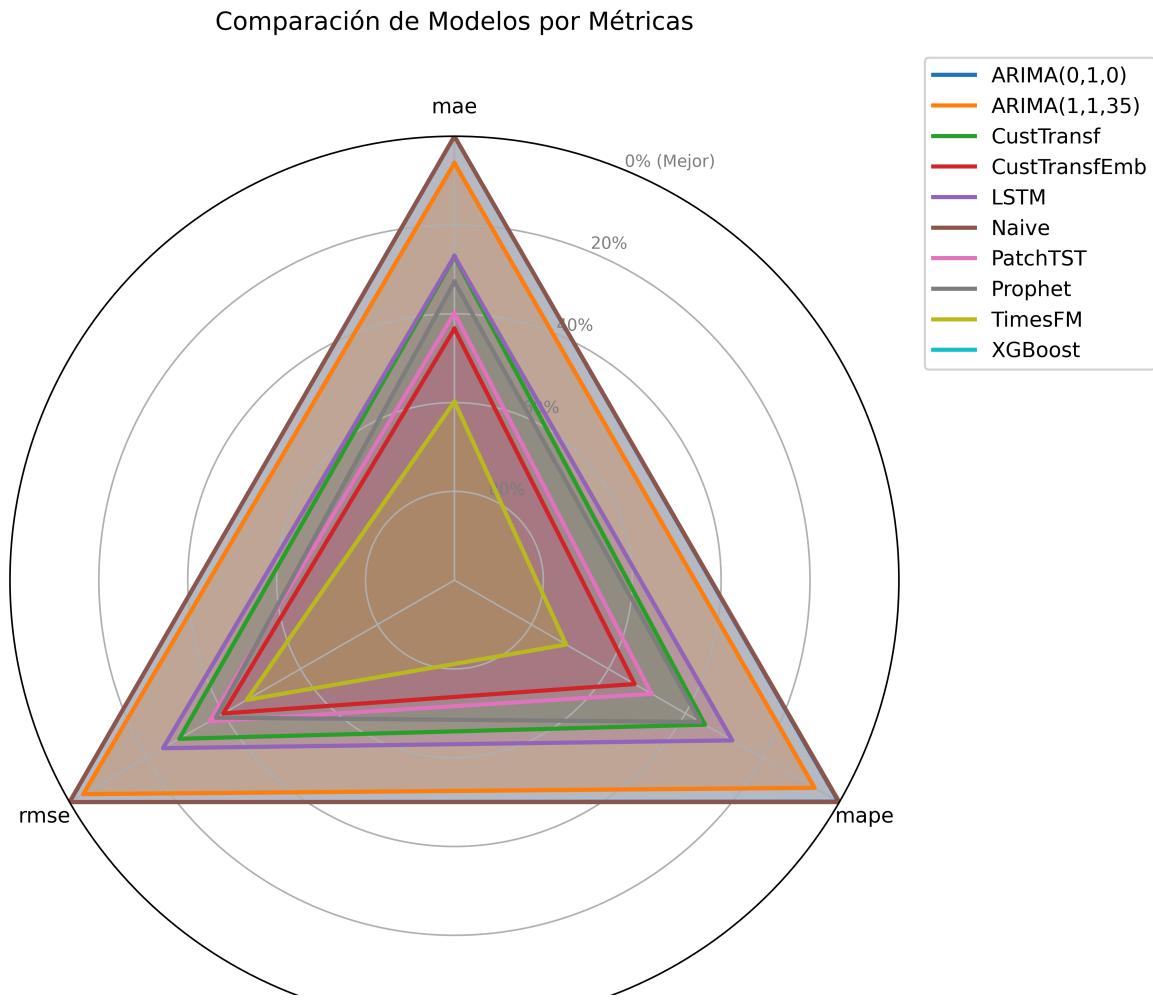
plt.tight_layout()
plt.savefig(filename, format='jpeg', dpi=600)
plt.show()

```

In [57]: `plot_metrics_comparison_bar(df_metrics, "resources/metrics_comparison.jpeg")`



In [58]: `plot_metrics_radial(df_metrics, "resources/metrics_radial.jpeg")`



5.1 Conclusiones

En este trabajo se exploró el uso de *Transformers* para series de tiempo, brindando una leve introducción a la arquitectura y su aplicación en el campo de series financieras.

Se implementaron dos modelos utilizando *Keras*, uno básico y otro con una codificación posicional alternativa, logrando resultados satisfactorios en la predicción de series temporales financieras (similares a modelos clásicos como LSTM o GRU).

También se exploró el uso de modelos pre-entrenados de *HuggingFace*, como **TimesFM** (originalmente de Google), que permite una mayor flexibilidad y adaptabilidad a diferentes series temporales, logrando resultados "interesantes" en la predicción de series temporales financieras.

Como conclusiones finales, se puede destacar que:

- 💡 **El entrenamiento de los modelos "custom" desde cero fue más efectivo que el modelo preentrenado** → Esto puede deberse a que **TimesFM** fue entrenado con varios tipos de datos, pero ninguno relacionado a las series financieras. Sin dudas, es evidencia que para las series de tiempo, podría ser muy importante hacer *fine-tuning* de los modelos con varias series del dominio.

- 💡 **El uso de codificación posicional alternativa mejora la capacidad del modelo para capturar patrones temporales** → Esto se debe a que la codificación posicional permite al modelo aprender patrones temporales de manera más efectiva, lo que mejora la precisión de las predicciones
- 💡 **El uso de modelos pre-entrenados puede ser útil para tareas específicas, pero no siempre es la mejor opción** → Esto se debe a que los modelos pre-entrenados pueden no estar adaptados a las características específicas de los datos, lo que puede afectar la precisión de las predicciones.
- 💡 **El uso de *Transformers* para series de tiempo es una línea de investigación prometedora** → Esto se debe a que los *Transformers* ofrecen ventajas significativas en términos de paralelización, atención y escalabilidad, lo que los hace adecuados para una amplia gama de tareas de series de tiempo, como se comentó al principio. Sin embargo, para este caso, sigue siendo el "mejor" modelo, simplemente predecir lo del día anterior.
- 💡 **Clara diferencia de performance en series temporales a diferencia de NLP** → Hay un clara diferencia de performance para estas arquitecturas en dominios del lenguaje natural con respecto al dominio de series temporales. Esto puede deberse a que en estos casos, donde los datos son escasos (solamente se intentó predecir con 1 año de datos), modelos con *high inductive bias* funcionan mejor que modelos con *low inductive bias*. Esto lo podemos evidenciar en que modelos clásicos como ARIMA han dado mejores resultados.
- 💡 **El uso de *Transformers* para series de tiempo también requiere de muchos datos** → Con los resultados, pudimos observar que se necesitan muchos más datos para que mejore la performance del modelo (relacionado al comentario anterior).

5.2 Mejoras y futuras líneas de investigación

Entre las mejoras y futuras líneas de investigación, se pueden destacar las siguientes:

- 💡 **Optimización de parámetros y arquitectura:** Sin duda un punto fuerte de este trabajo la no optimización de la arquitectura e hiperparámetros. Esto se debe a una restricción temporal, pero sería interesante realizar este proceso para los modelos "custom".
- 💡 **Realizar _fine-tuning_ sobre el modelo TimesFM :** Creemos que es muy importante para estos casos que los modelos estén pre-entrenados con datos relacionados. Es por esto, que se sugiere probar realizar *fine-tuning* sobre este modelo con series financieras similares.
- 💡 **Probar con otros modelos pre-entrenados:** Existen otros modelos pre-entrenados que podrían ser útiles para este caso, como LogTrans o Informer, que podrían mejorar la precisión de las predicciones.
- 💡 **Aumentar la cantidad de datos:** Sin dudas mejoraría la performance de estas arquitecturas tener más datos, como en NLP. En este ejemplo se evidenció que el Transformer no logra generalizar correctamente con esta cantidad de datos.

- **💡 Explorar otras características y embeddings:** En este caso, se mejoró con la incorporación de *embeddings*. Esto podría dar lugar a explorar esta línea de mejor generación de *embeddings* como incorporar más características a la serie.
- **💡 Tamaño de ventana:** En esta exploración se utilizó simplemente un tamaño de ventana chico para continuar con el análisis previo pero se podría incluir en la optimización de hiperparámetros o investigar más su inferencia.
- **💡 Serie univariable en PatchTST:** La forma de implementar PatchTST en el ejemplo no es del todo alineada con el paper, dado que este propone utilizar varias series temporales y en este caso se utilizó solamente una (la de Bitcoin). Pero lo ideal sería realizar un análisis con varias series en varios canales.

6. Referencias

- Transformers in time series: A survey | <https://arxiv.org/abs/2202.07125>
- Transformer models: an introduction and catalog | <https://arxiv.org/abs/2302.07730>
- Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting (2020) | <https://arxiv.org/abs/1907.00235>
- How to apply transformers to time series models | <https://medium.com/intel-tech/how-to-apply-transformers-to-time-series-models-spacetimeformer-e452f2825d2e>
- A Time Series is Worth 64 Words: Long-term Forecasting with Transformers (2023) | <https://arxiv.org/abs/2211.14730>
- A decoder-only foundation model for time-series forecasting | <https://huggingface.co/papers/2310.10688>
- LogTrans: Providing Efficient Local-Global Fusion with Transformer and CNN Parallel Network for Biomedical Image Segmentation | <https://ieeexplore.ieee.org/document/10074688>
- Pyraformer: Low-Complexity Pyramidal Attention for Long-Range Time Series Modeling and Forecasting | <https://openreview.net/forum?id=0EXmFzUn5I>
- FEDformer: Frequency Enhanced Decomposed Transformer for Long-term Series Forecasting | <https://arxiv.org/abs/2201.12740>
- Is my take on transformers in time series reasonable / where is it wrong? |https://www.reddit.com/r/MachineLearning/comments/1k63r4a/d_is_my_take_on_transformer/

