

# LSTM-Based Time Series Prediction for Fuel Consumption in Guatemala

Este notebook presenta un análisis exhaustivo del desempeño de dos modelos de pronóstico: LSTM (Long Short-Term Memory) y ARIMA (Autoregressive Integrated Moving Average), aplicados a los conjuntos de datos relacionados con el consumo y costo de diésel, así como la importación regular de combustible en Guatemala.

A través de este análisis, se busca no solo entender el comportamiento de los modelos, sino también proporcionar información valiosa sobre las tendencias y patrones en el consumo y costo de diésel en Guatemala y la importación del combustible regular.

## Authors:

- [Andrea Ramírez](#)
- [Adrian Flores](#)

---

## Import Libraries

```
In [ ]: #!/pip install keras-tuner
```

```
In [ ]: # Data manipulation and visualization
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from statsmodels.tsa.stattools import adfuller
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error
import tensorflow as tf
from tensorflow import keras
import keras_tuner as kt
from operator import concat

# Standard libraries
import warnings
warnings.filterwarnings('ignore')

# ===== Reproducibility Seed =====
# Set a fixed seed for the random number generator for reproducibility
random_state = 42

# Set matplotlib inline
%matplotlib inline
```

```
# Set default figure size
plt.rcParams['figure.figsize'] = (6, 4)

# Define custom color palette
palette = sns.color_palette("viridis", 12)

# Set the style of seaborn
sns.set(style="whitegrid")

# Set TensorFlow Global Seed
tf.random.set_seed(random_state)
```

## Data Upload

```
In [ ]: def read_and_process_excel(file_names):
        dfs = [] # Initialize an empty list to store DataFrames

        for file_name in file_names:
            # Read the Excel file while skipping the first six rows of headers
            df = pd.read_excel(file_name, skiprows=6)

            # Drop the last three rows from the DataFrame to remove any unwanted
            df = df.iloc[:-3]

            # Convert the 'Fecha' column to datetime format
            df['Fecha'] = pd.to_datetime(df['Fecha'])

            # Set the 'Fecha' column as the index of the DataFrame
            df.set_index('Fecha', inplace=True)

            # Select only the specified columns and create a new column 'Diesel'
            df['Diesel'] = df['Diesel alto azufre'].fillna(0) + df['Diesel bajo azufre']

            # Select only the relevant columns: Gasolina regular, Gasolina superior, Gas licuado de pe
            df = df[['Gasolina regular', 'Gasolina superior', 'Gas licuado de pe

            # Append the processed DataFrame to the list
            dfs.append(df)

        return dfs # Return the list of DataFrames
```

```
In [ ]: # List of Excel file names to be processed
file_names = ["consumo.xlsx", "importacion.xlsx"]
dataset_names = ["Consumo", "Importacion", "Precios"]

# Call the function to read and process the Excel files, storing the result
dataframes = read_and_process_excel(file_names)
```

```
In [ ]: def read_price_df(sheetname, skip):
        # Read the Excel file while skipping the first six rows of headers
        df = pd.read_excel("precios.xlsx", skiprows=skip, sheet_name=sheetname)
        # Drop the last three rows from the DataFrame to remove any unwanted data
        df = df.iloc[1:-3]
        # Convert the 'Fecha' column to datetime format
```

```

df['Fecha'] = pd.to_datetime(df['FECHA'])
# Set the 'Fecha' column as the index of the DataFrame
df.set_index('Fecha', inplace=True)
# Remove last column
df = df.iloc[:, :-1]
# Rename the columns correctly
df.rename(columns={
    'FECHA': 'Fecha',
    'Tipo de Cambio': 'Tipo de Cambio',
    'Superior': 'Gasolina superior',
    'Regular': 'Gasolina regular',
    'Diesel': 'Diesel',
    'Bunker': 'Bunker',
    'Glp Cilindro 25Lbs.': 'Gas licuado de petróleo'
}, inplace=True)
# Select only the relevant columns: Gasolina regular, Gasolina superior, Gas
df = df[['Gasolina regular', 'Gasolina superior', 'Gas licuado de petróleo']]
# Drop NaN values from the final DataFrame
df.dropna(inplace=True)
return df

```

```

In [ ]: list_price_params = [("2021", 6), ("2022", 6), ("2023", 7), ("2024", 7)]

# Initialize an empty list to hold DataFrames
df_list = []

# Loop through each parameter to read and append DataFrames to the list
for year, skip in list_price_params:
    df = read_price_df(year, skip)
    df_list.append(df)

# Concatenate all DataFrames in the list into a single DataFrame
df = pd.concat(df_list)
# Optionally, sort the index if necessary
df.sort_index(inplace=True)
# Display the final DataFrame
dataframes.append(df)

```

## Exploratory Analysis

### (1) Exploración y Limpieza Inicial de los Datos

Para facilitar la comprensión y el manejo del conjunto de datos, se procederá a modificar los nombres de las variables. Este cambio permitirá una organización más clara y una interpretación más precisa de la información.

```

In [ ]: # Dictionary to rename columns for better readability
rename_col = {
    'Gasolina regular': 'gasoline_regular', # Renaming 'Gasolina regular'
    'Gasolina superior': 'gasoline_superior', # Renaming 'Gasolina superior'
    'Gas licuado de petróleo': 'liquefied_gas', # Renaming 'Gas licuado de petróleo'
    'Diesel': 'diesel' # Renaming 'Diesel' to 'diesel'
}

```

}

```
In [ ]: for i, df in enumerate(dataframes):
        # Use a pandas function to rename the current function
        df = df.rename(columns = rename_col)
        # Change the index name from 'Fecha' to 'date'
        df.rename_axis('date', inplace=True)
        # Ensure all columns are numeric
        df = df.astype('float64')
        # Save changes
        dataframes[i] = df
        print(df.head(2), "\n")
```

	gasoline_regular	gasoline_superior	liquefied_gas	diesel
date				
2000-01-01	202645.20	308156.82	194410.476190	634667.06
2000-02-01	205530.96	307766.31	174710.552381	642380.66

	gasoline_regular	gasoline_superior	liquefied_gas	diesel
date				
2001-01-01	177776.50	373963.96	194065.738095	566101.99
2001-02-01	123115.99	243091.07	170703.380952	489525.80

	gasoline_regular	gasoline_superior	liquefied_gas	diesel
date				
2021-01-01	21.11	21.91	99.0	17.61
2021-01-02	21.11	21.91	99.0	17.61

## Time Series Forecasting

### (1) Elección de Series de Tiempo a Utilizar

- **Consumo de Diésel:** Para analizar las tendencias y patrones en el uso de diésel a lo largo de los años, dadas las fuertes fluctuaciones observadas.
- **Importación de Gasolina Regular:** Para obtener información sobre el aparente aumento en la importación de la gasolina regular.
- **Precio de Diesel:** Para analizar las tendencias en el precio de los combustibles a lo largo de los años, dadas sus fluctuaciones significativas.

```
In [ ]: diesel_consumption = dataframes[0].diesel
        regular_importation = dataframes[1].gasoline_regular
        diesel_price = dataframes[2].diesel
```

### (2) Información General de las Series de Tiempo

```
In [ ]: def get_information(df):
        # Find the start of the time series
```

```
start_date = df.index.min()
# Find the end of the time series
end_date = df.index.max()
# Find the frequency of the time series
frequency = pd.infer_freq(df.index)

# Print the results nicely
print("Time Series Analysis:")
print("-----")
print(f"Start Date: {start_date.date()}")
print(f"End Date: {end_date.date()}")
print(f"Frequency: {frequency}")
```

## (1) Consumo de Diésel

```
In [ ]: get_information(diesel_consumption)
```

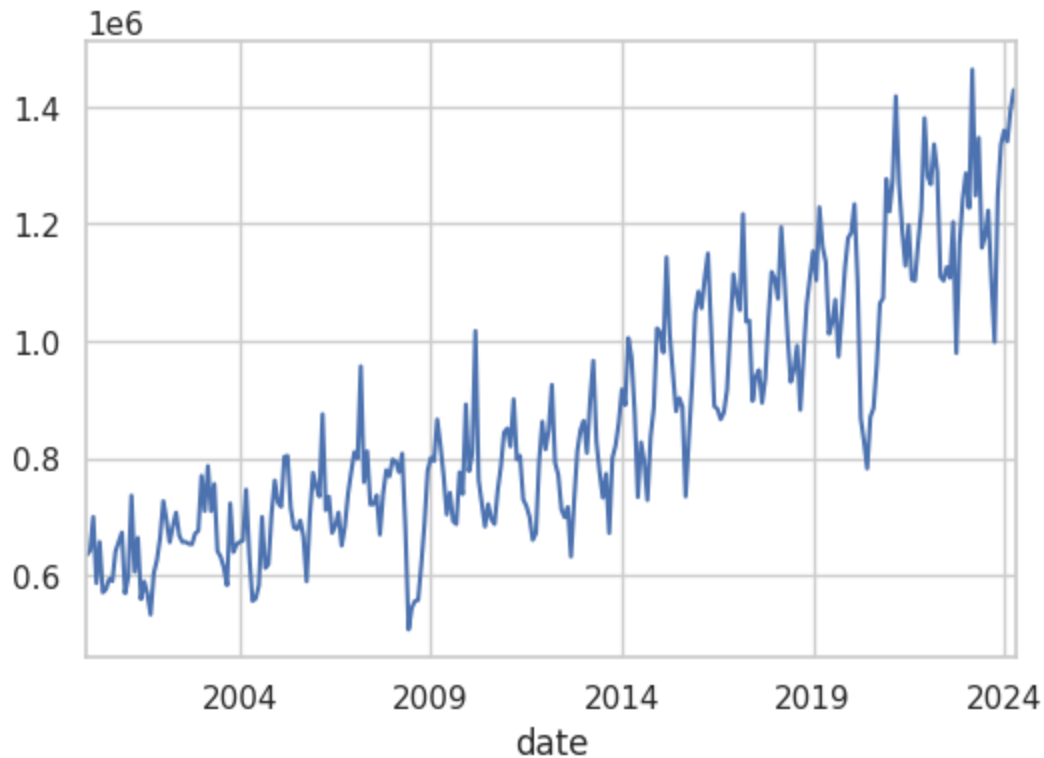
```
Time Series Analysis:
-----
Start Date: 2000-01-01
End Date: 2024-05-01
Frequency: MS
```

### Observaciones 💡 -->

- **Fecha de inicio:** 2000-01-01. Indica que la serie de tiempo comienza el 1 de enero de 2000.
- **Fecha de finalización:** 2024-05-01. Indica que la serie de tiempo se extiende hasta el 1 de mayo de 2024.
- **Frecuencia:** MS (Inicio del mes). Significa que los datos se registran al inicio de cada mes.

```
In [ ]: diesel_consumption.plot()
```

```
Out[ ]: <Axes: xlabel='date'>
```



## (2) Importación de Gasolina Regular

```
In [ ]: get_information(regular_importation)
```

Time Series Analysis:

-----

Start Date: 2001-01-01

End Date: 2024-05-01

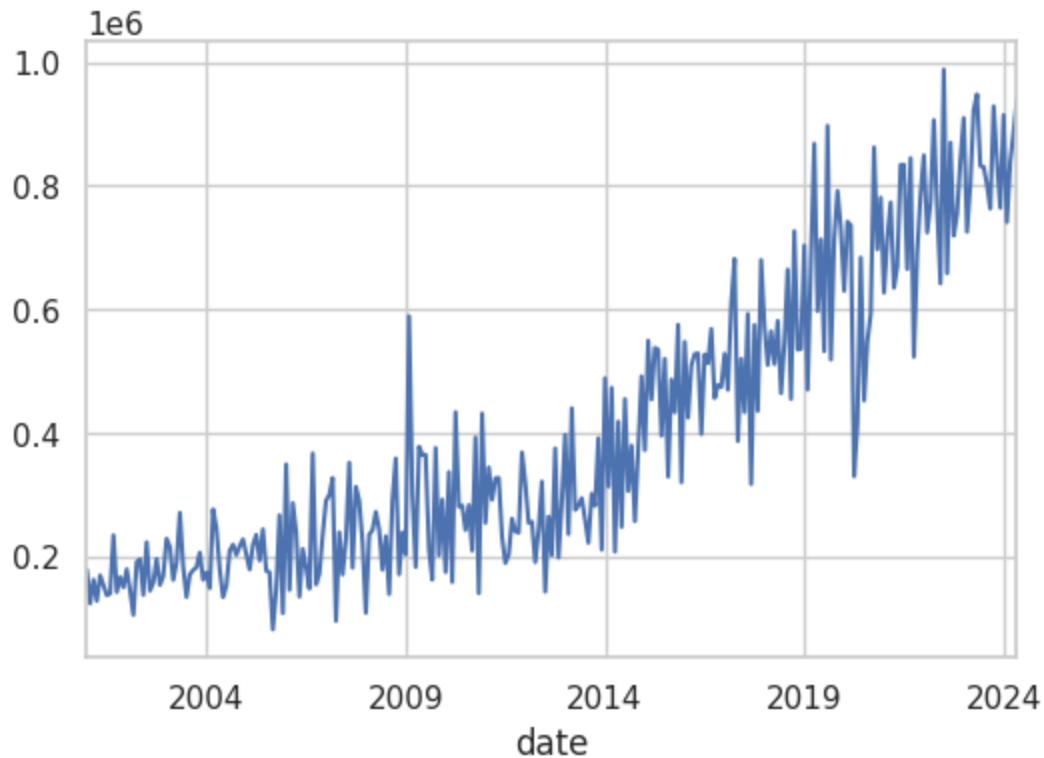
Frequency: MS

### Observaciones 💡 -->

- **Fecha de inicio:** 2001-01-01. Indica que la serie de tiempo comienza el 1 de enero de 2001.
- **Fecha de finalización:** 2024-05-01. Indica que la serie de tiempo se extiende hasta el 1 de mayo de 2024.
- **Frecuencia:** MS (Inicio del mes). Significa que los datos se registran al inicio de cada mes.

```
In [ ]: regular_importation.plot()
```

```
Out[ ]: <Axes: xlabel='date'>
```



### (3) Precio del Diesel

```
In [ ]: get_information(diesel_price)
```

Time Series Analysis:

-----

Start Date: 2021-01-01

End Date: 2024-07-28

Frequency: None

#### Observaciones 💡 -->

- **Fecha de inicio:** 2021-01-01. Indica que la serie de tiempo comienza el 1 de enero de 2021.
- **Fecha de finalización:** 2024-07-28. Indica que la serie de tiempo se extiende hasta el 28 de julio de 2024.
- **Frecuencia:** Ninguna. Significa que no es posible inferir de manera automática la frecuencia de los datos. A pesar de esto, se sabe que se tiene una entrada por día hasta la última fecha registrada.

```
In [ ]: diesel_price.plot()
```

```
Out[ ]: <Axes: xlabel='date'>
```



### (3) ADF y Transformaciones

```
In [ ]: def adf_test(series):
    result = adfuller(series)
    adf_statistic = result[0]
    p_value = result[1]
    critical_values = result[4]

    print('ADF Statistic:', adf_statistic)
    print('p-value:', p_value)
    print('Critical Values:')
    for key, value in critical_values.items():
        print(f' {key}: {value:.3f}')

    if p_value < 0.05:
        print("The series is stationary (reject H0).")
        return True

    else:
        print("The series is non-stationary (fail to reject H0).")
        return False
```



```
In [ ]: def find_stationarity(series):  
        diff_series = series.copy()  
        differencing_order = 0  
  
        while not adf_test(diff_series):  
            differencing_order += 1  
            diff_series = diff_series.diff().dropna()  
  
        print(f"The series became stationary after {differencing_order} differenc  
        return diff_series
```

```
In [ ]: diesel_consumption_diff = find_stationarity(diesel_consumption)  
        regular_importation_diff = find_stationarity(regular_importation)  
        diesel_price_diff = find_stationarity(diesel_price)
```

```
ADF Statistic: 0.145239821178108
p-value: 0.9690175028779469
Critical Values:
  1%: -3.454
  5%: -2.872
 10%: -2.572
The series is non-stationary (fail to reject H0).
ADF Statistic: -7.094677414843632
p-value: 4.3199961290916e-10
Critical Values:
  1%: -3.454
  5%: -2.872
 10%: -2.572
The series is stationary (reject H0).
The series became stationary after 1 differencings.
```

```
ADF Statistic: 0.8288289928204887
p-value: 0.992087554110633
Critical Values:
  1%: -3.455
  5%: -2.872
 10%: -2.572
The series is non-stationary (fail to reject H0).
ADF Statistic: -10.111097641346007
p-value: 9.998584727414137e-18
Critical Values:
  1%: -3.455
  5%: -2.872
 10%: -2.572
The series is stationary (reject H0).
The series became stationary after 1 differencings.
```

```
ADF Statistic: -2.377813354587725
p-value: 0.14809658351111532
Critical Values:
  1%: -3.435
  5%: -2.864
 10%: -2.568
The series is non-stationary (fail to reject H0).
ADF Statistic: -7.090525705456006
p-value: 4.4224777713128013e-10
Critical Values:
  1%: -3.435
  5%: -2.864
 10%: -2.568
The series is stationary (reject H0).
The series became stationary after 1 differencings.
```

## (4) LSTM Models

### (1) Normalización de Datos

```
In [ ]: # Create an instance of StandardScaler for scaling time series data
        consumption_scaler = StandardScaler()
```

```

importation_scaler = StandardScaler()
price_scaler = StandardScaler()

def scale_time_series(series, scaler):
    # Reshape the series to 2D
    series_resaped = series.values.reshape(-1, 1)
    scaled_values = scaler.fit_transform(series_resaped)
    print(scaled_values[1:5], "\n")
    return scaled_values

```

```

In [ ]: diesel_consumption_scaled = scale_time_series(diesel_consumption_diff, consu
regular_importation_scaled = scale_time_series(regular_importation_diff, imp
diesel_price_scaled = scale_time_series(diesel_price_diff, price_scaler)

```

```

[[ 0.6596291 ]
 [-1.39176556]
 [ 0.81270602]
 [-1.0670791 ]]

```

```

[[ 0.26658337]
 [-0.27563983]
 [ 0.28724053]
 [-0.13780327]]

```

```

[[-0.01703256]
 [-0.01703256]
 [-0.01703256]
 [-0.01703256]]

```

## (2) División de Datos: Entrenamiento, Validación y Prueba

```

In [ ]: def train_val_test_split(data_array, train_size=0.85, val_size=0.075, test_s
    # Calculate the number of entries in the dataset
    n = len(data_array)

    # Ensure that the sizes sum to 1
    assert np.isclose(train_size + val_size + test_size, 1.0), "Sizes must s

    # Calculate the indices for the splits
    train_end = int(train_size * n) # End index for the training set
    val_end = train_end + int(val_size * n) # End index for the validation

    # Split the dataset into train, validation, and test sets
    train_array = data_array[:train_end]
    val_array = data_array[train_end:val_end]
    test_array = data_array[val_end:]

    return train_array, val_array, test_array

```

```

In [ ]: # Diesel Consumption Data
train_consumption, val_consumption, test_consumption = train_val_test_split(
# Regular Fuel Importation Data
train_regular, val_regular, test_regular = train_val_test_split(regular_imp
# Diesel Price Data

```

```
train_price, val_price, test_price = train_val_test_split(diesel_price_scale
```

### (3) Transformaciones a los Conjuntos de Datos

```
In [ ]: def supervised_series(series, lags=1):
    # Initialize lists to hold input and output sequences
    input_series = []
    output_series = []
    # Loop through the series to create input-output pairs
    for i in range(len(series) - lags):
        # Collect the input sequence (previous 'lags' observations)
        input_value = series[i:(i + lags), 0]
        # Collect the output value (next observation)
        output_value = series[i + lags, 0]
        # Append to the respective lists
        input_series.append(input_value)
        output_series.append(output_value)
    return np.array(input_series), np.array(output_series)
```

```
In [ ]: # Diesel Consumption Data
x_train_consumption, y_train_consumption = supervised_series(train_consumption, lags=1)
x_val_consumption, y_val_consumption = supervised_series(val_consumption, lags=1)
x_test_consumption, y_test_consumption = supervised_series(test_consumption, lags=1)

# Regular Fuel Importation Data
x_train_regular, y_train_regular = supervised_series(train_regular, lags=1)
x_val_regular, y_val_regular = supervised_series(val_regular, lags=1)
x_test_regular, y_test_regular = supervised_series(test_regular, lags=1)

# Diesel Price Data
x_train_price, y_train_price = supervised_series(train_price, lags=1)
x_val_price, y_val_price = supervised_series(val_price, lags=1)
x_test_price, y_test_price = supervised_series(test_price, lags=1)
```

```
In [ ]: # Diesel Consumption Data
x_train_consumption = np.reshape(x_train_consumption, (x_train_consumption.shape[0], 1, 1))
x_val_consumption = np.reshape(x_val_consumption, (x_val_consumption.shape[0], 1, 1))
x_test_consumption = np.reshape(x_test_consumption, (x_test_consumption.shape[0], 1, 1))

# Regular Fuel Importation Data
x_train_regular = np.reshape(x_train_regular, (x_train_regular.shape[0], 1, 1))
x_val_regular = np.reshape(x_val_regular, (x_val_regular.shape[0], 1, 1))
x_test_regular = np.reshape(x_test_regular, (x_test_regular.shape[0], 1, 1))

# Diesel Price Data
x_train_price = np.reshape(x_train_price, (x_train_price.shape[0], 1, 1))
x_val_price = np.reshape(x_val_price, (x_val_price.shape[0], 1, 1))
x_test_price = np.reshape(x_test_price, (x_test_price.shape[0], 1, 1))
```

```
In [ ]: consumption_dataset = {
    "x_train": x_train_consumption,
    "y_train": y_train_consumption,
    "x_val": x_val_consumption,
    "y_val": y_val_consumption,
```

```

    "x_test": x_test_consumption,
    "y_test": y_test_consumption
}

importation_dataset = {
    "x_train": x_train_regular,
    "y_train": y_train_regular,
    "x_val": x_val_regular,
    "y_val": y_val_regular,
    "x_test": x_test_regular,
    "y_test": y_test_regular
}

price_dataset = {
    "x_train": x_train_price,
    "y_train": y_train_price,
    "x_val": x_val_price,
    "y_val": y_val_price,
    "x_test": x_test_price,
    "y_test": y_test_price
}

```

#### (4) Creación de Modelos LSTM

Para las primeras iteraciones de cada serie, se crearán modelos con dos atributos distintos: cantidad de unidades en la capa LSTM y optimizador. En esta iteración, la elección de estos atributos es arbitraria y se detalla a continuación:

- **Primer Modelo:**

- Unidades LSTM: 1
- Optimizador: Adam (LR = 0.001)

- **Segundo Modelo:**

- Unidades LSTM: 5
- Optimizador: SGD (LR = 0.001)

```

In [ ]: def init_model(param_dict, name=None):
    inputs = keras.layers.Input((param_dict["step"], param_dict["features"]))
    lstm_out = keras.layers.LSTM(param_dict["units"])(inputs)
    outputs = keras.layers.Dense(1)(lstm_out)

    model = keras.Model(inputs=inputs, outputs=outputs)
    if name:
        model.name = name
    model.summary()

    model.compile(loss='mean_squared_error', optimizer=param_dict["optimizer"])

    return model

```

```

In [ ]: def fit_model(model, dataset_dict, epochs, batch):
    history = model.fit(

```

```

        x = dataset_dict["x_train"],
        y = dataset_dict["y_train"],
        batch_size = batch,
        epochs = epochs,
        shuffle = False,
        validation_data = (dataset_dict["x_val"], dataset_dict["y_val"]),
        verbose=0
    )

    print("Training Loss")
    model.evaluate(
        x = dataset_dict["x_train"],
        y = dataset_dict["y_train"]
    )
    print("\nValidation Loss")
    model.evaluate(
        x = dataset_dict["x_val"],
        y = dataset_dict["y_val"]
    )
    print("\nTesting Loss")
    model.evaluate(
        x = dataset_dict["x_test"],
        y = dataset_dict["y_test"]
    )

    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='center')

```

```

In [ ]: first_params = {
        "batch": 1,
        "units": 1,
        "step": 1,
        "features": 1,
        "optimizer": "adam",
    }

    second_params = {
        "batch": 1,
        "units": 5,
        "step": 1,
        "features": 1,
        "optimizer": "sgd",
    }

```

### (1) Primer Modelo

```

In [ ]: consumption_model = init_model(first_params, "consumption_model")
        regular_model = init_model(first_params, "importation_model")
        price_model = init_model(first_params, "price_model")

```

**Model: "consumption\_model"**

Layer (type)	Output Shape
input_layer ( <a href="#">InputLayer</a> )	( <a href="#">None</a> , 1, 1)
lstm ( <a href="#">LSTM</a> )	( <a href="#">None</a> , 1)
dense ( <a href="#">Dense</a> )	( <a href="#">None</a> , 1)

**Total params:** 14 (56.00 B)

**Trainable params:** 14 (56.00 B)

**Non-trainable params:** 0 (0.00 B)

**Model:** "importation\_model"

Layer (type)	Output Shape
input_layer_1 ( <a href="#">InputLayer</a> )	( <a href="#">None</a> , 1, 1)
lstm_1 ( <a href="#">LSTM</a> )	( <a href="#">None</a> , 1)
dense_1 ( <a href="#">Dense</a> )	( <a href="#">None</a> , 1)

**Total params:** 14 (56.00 B)

**Trainable params:** 14 (56.00 B)

**Non-trainable params:** 0 (0.00 B)

**Model:** "price\_model"

Layer (type)	Output Shape
input_layer_2 ( <a href="#">InputLayer</a> )	( <a href="#">None</a> , 1, 1)
lstm_2 ( <a href="#">LSTM</a> )	( <a href="#">None</a> , 1)
dense_2 ( <a href="#">Dense</a> )	( <a href="#">None</a> , 1)


**Total params:** 14 (56.00 B)

**Trainable params:** 14 (56.00 B)

**Non-trainable params:** 0 (0.00 B)

```
In [ ]: fit_model(consumption_model, consumption_dataset, 50, first_params["batch"])
```

Training Loss

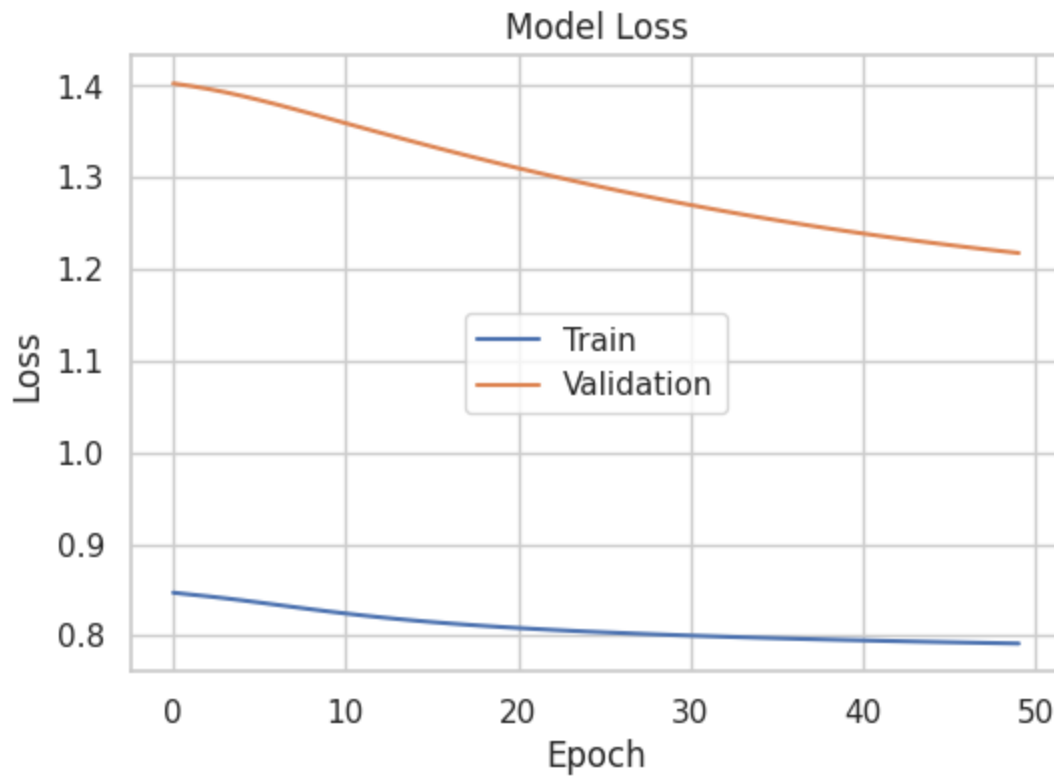
8/8  0s 3ms/step - loss: 0.6466

Validation Loss

1/1  0s 25ms/step - loss: 1.2176

Testing Loss

1/1  0s 24ms/step - loss: 2.2250



```
In [ ]: fit_model(regular_model, importation_dataset, 50, first_params["batch"])
```

Training Loss

8/8 ████████████████████ 0s 2ms/step - loss: 0.3954

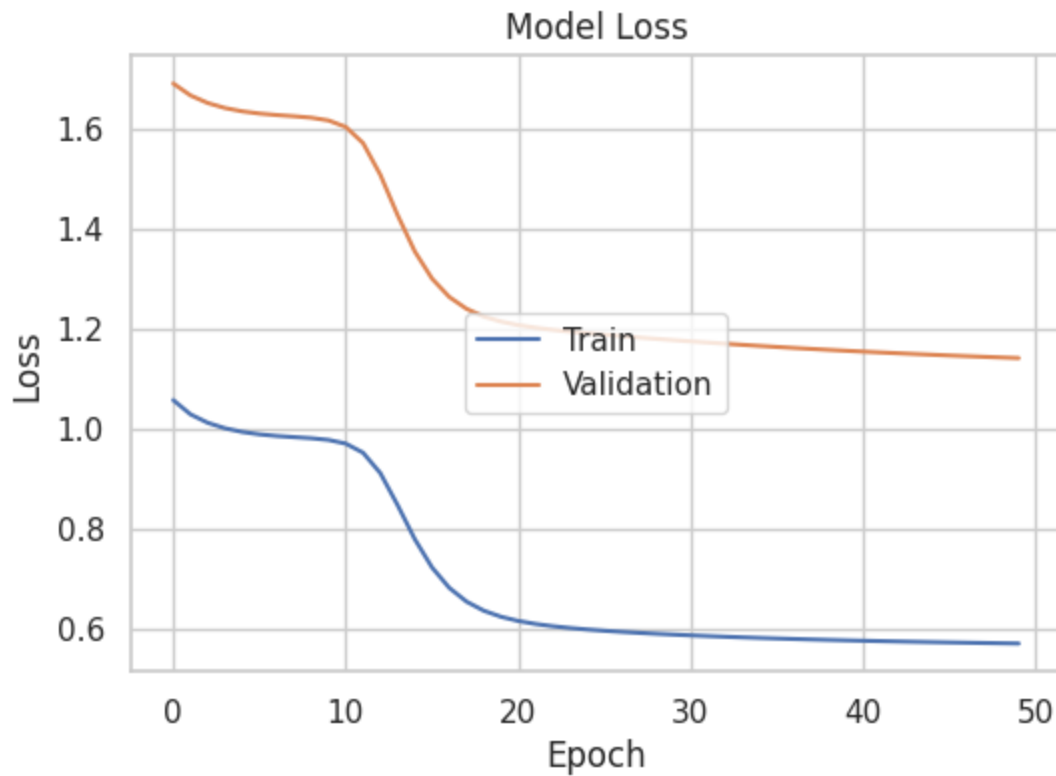
Validation Loss

1/1 ████████████████████ 0s 27ms/step - loss: 1.1404

Testing Loss

1/1 ████████████████████ 0s 28ms/step - loss: 0.5667





```
In [ ]: fit_model(price_model, price_dataset, 50, first_params["batch"])
```

Training Loss

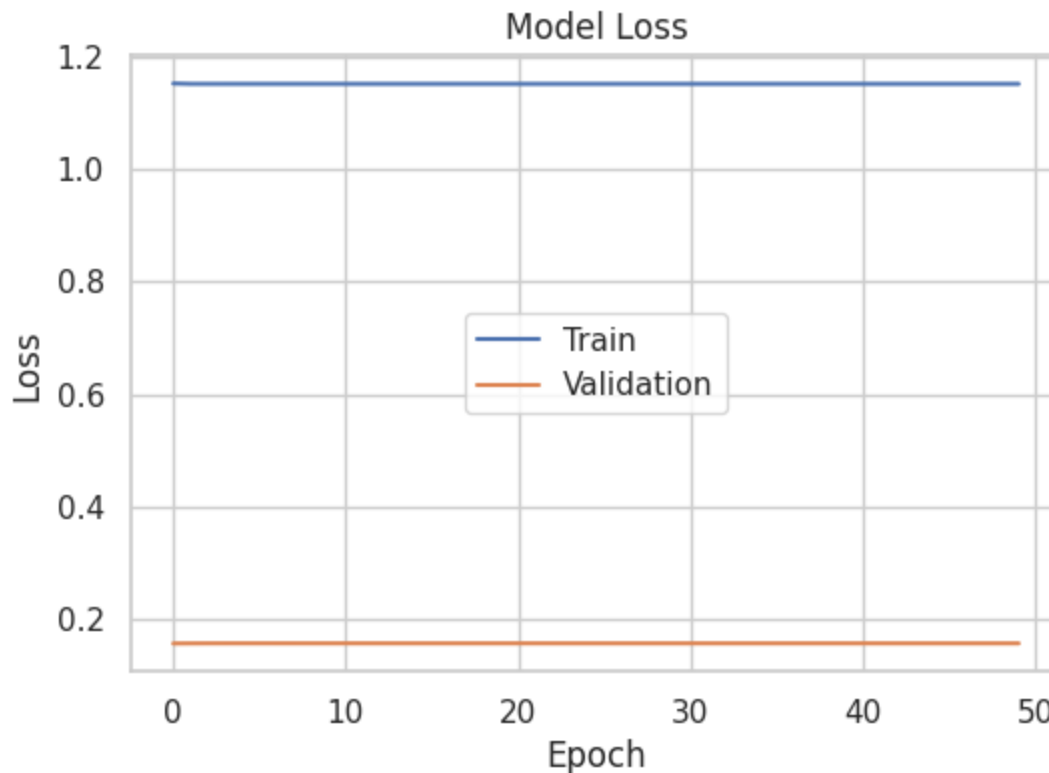
35/35 ████████████████████ 0s 1ms/step - loss: 0.8831

Validation Loss

3/3 ████████████████████ 0s 4ms/step - loss: 0.1934

Testing Loss

4/4 ████████████████████ 0s 3ms/step - loss: 0.1265



#### Observaciones 💡 -->

- Los modelos de importación y consumo no muestran señales de sobreajuste, ya que sus curvas de entrenamiento y validación no convergen en ningún momento.
- El modelo de precios presenta una pérdida casi constante a lo largo de todas las épocas, con una menor pérdida en validación que en entrenamiento. Este comportamiento podría deberse a una mala configuración de los parámetros del modelo o a problemas con las transformaciones de datos previas.
- El modelo de importación tiene la curva de pérdida más definida y, al mismo tiempo, la pérdida más alta entre todos los modelos. Por otro lado, el modelo de consumo no muestra una reducción significativa en la pérdida, y la pérdida del modelo de precios permanece constante durante las 50 épocas.
- Es probable que no se necesiten más épocas en iteraciones futuras, ya que los modelos convergen a un valor de pérdida antes de alcanzar las 50 épocas.

#### (2) Segundo Modelo

```
In [ ]: consumption_model = init_model(second_params, "consumption_model_2")
        regular_model = init_model(second_params, "importation_model_2")
        price_model = init_model(second_params, "price_model_2")
```

Model: "consumption\_model\_2"

Layer (type)	Output Shape
input_layer_3 (InputLayer)	(None, 1, 1)
lstm_3 (LSTM)	(None, 5)
dense_3 (Dense)	(None, 1)

**Total params:** 146 (584.00 B)

**Trainable params:** 146 (584.00 B)

**Non-trainable params:** 0 (0.00 B)

**Model:** "importation\_model\_2"

Layer (type)	Output Shape
input_layer_4 (InputLayer)	(None, 1, 1)
lstm_4 (LSTM)	(None, 5)
dense_4 (Dense)	(None, 1)

**Total params:** 146 (584.00 B)

**Trainable params:** 146 (584.00 B)

**Non-trainable params:** 0 (0.00 B)

**Model:** "price\_model\_2"

Layer (type)	Output Shape
input_layer_5 (InputLayer)	(None, 1, 1)
lstm_5 (LSTM)	(None, 5)
dense_5 (Dense)	(None, 1)


**Total params:** 146 (584.00 B)

**Trainable params:** 146 (584.00 B)

**Non-trainable params:** 0 (0.00 B)

```
In [ ]: fit_model(consumption_model, consumption_dataset, 50, second_params["batch"])
```

Training Loss

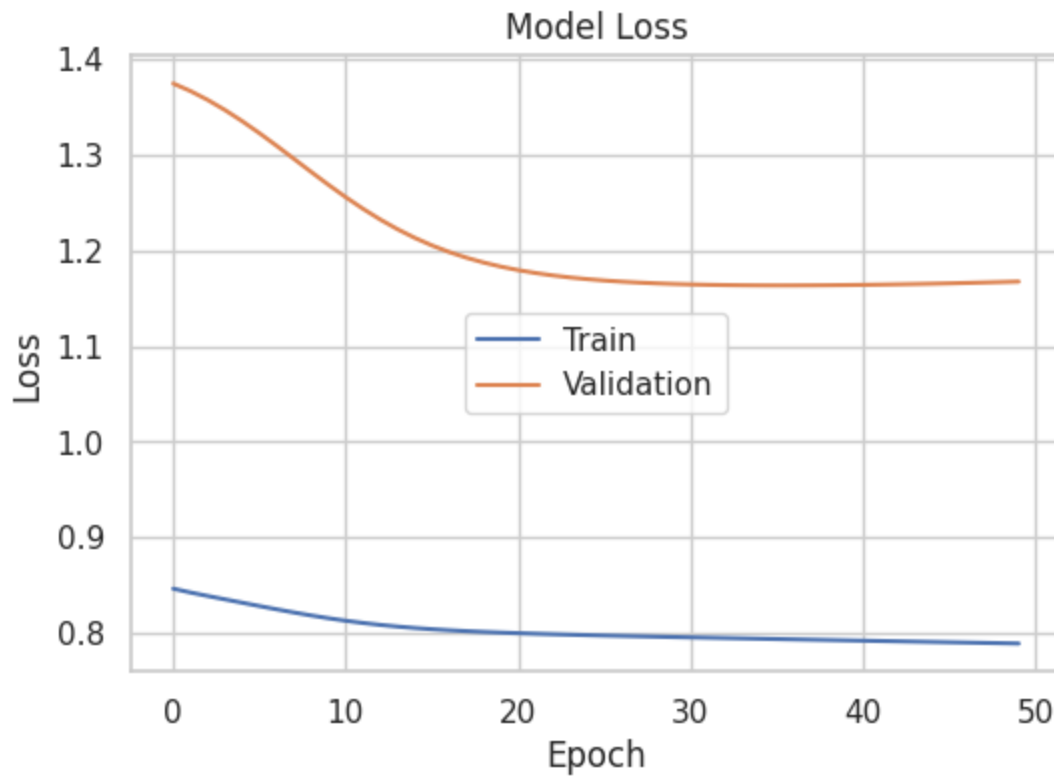
8/8  0s 3ms/step - loss: 0.6398

Validation Loss

1/1  0s 39ms/step - loss: 1.1678

Testing Loss

1/1  0s 55ms/step - loss: 2.5892



```
In [ ]: fit_model(regular_model, importation_dataset, 50, second_params["batch"])
```

Training Loss

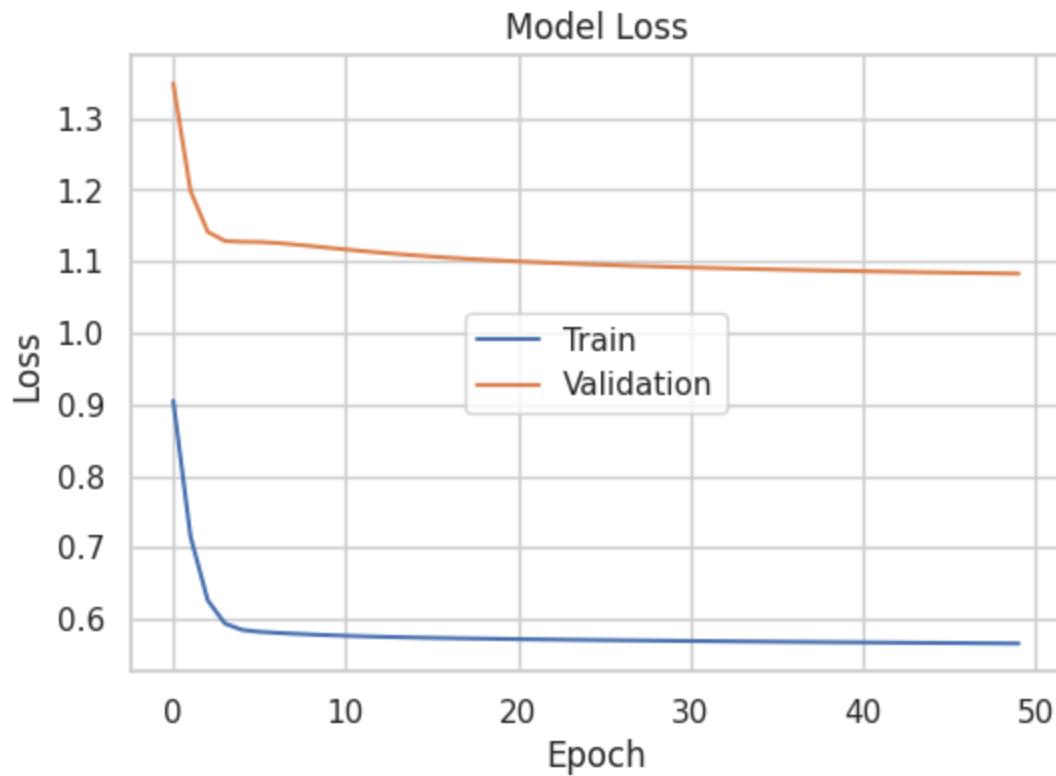
8/8 ████████████████████ 0s 2ms/step - loss: 0.3907

Validation Loss

1/1 ████████████████████ 0s 27ms/step - loss: 1.0829

Testing Loss

1/1 ████████████████████ 0s 25ms/step - loss: 0.5477



```
In [ ]: fit_model(price_model, price_dataset, 50, second_params["batch"])
```

Training Loss

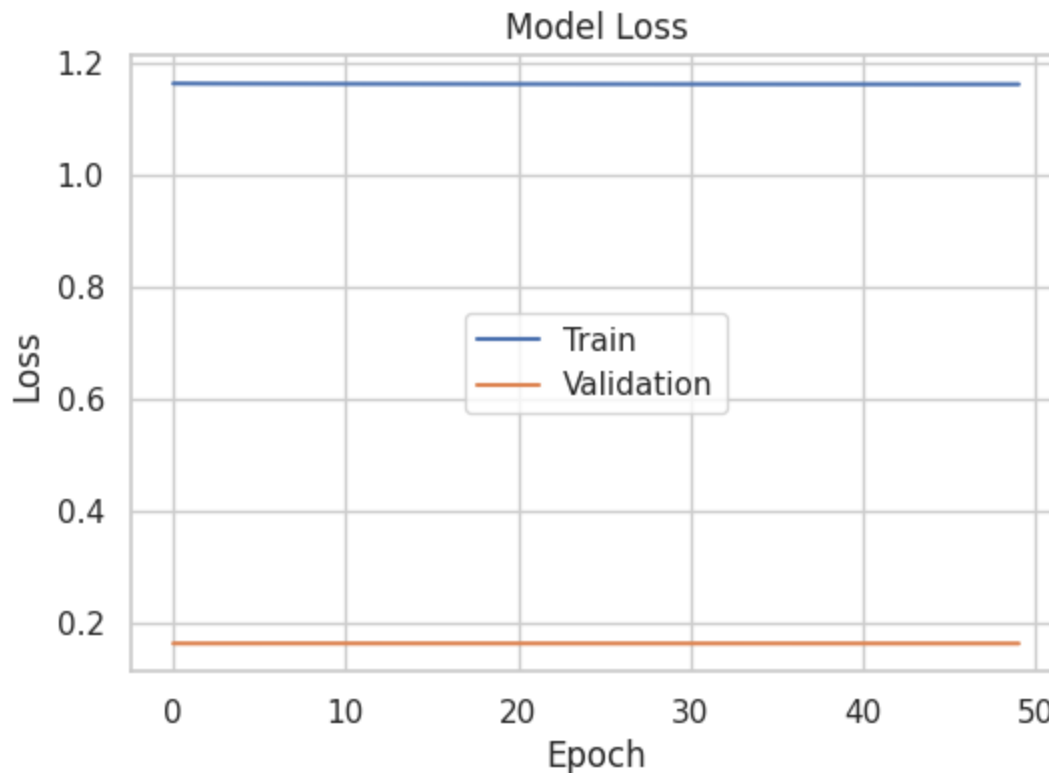
35/35  0s 1ms/step - loss: 0.8914

Validation Loss

3/3  0s 4ms/step - loss: 0.2024

Testing Loss

4/4  0s 3ms/step - loss: 0.1191



#### Observaciones 💡 -->

- Al cambiar el optimizador y la cantidad de unidades en la capa LSTM, se observa un comportamiento ligeramente distinto en las gráficas de pérdida de los modelos. El modelo de precios mostró el mismo comportamiento que en su versión anterior, mientras que los modelos de consumo e importación presentan curvas menos pronunciadas.
- El modelo de consumo mostró un comportamiento casi constante, con una diferencia de aproximadamente 0.05 entre la pérdida de la primera época y la última.
- El modelo de importación sigue registrando los valores de pérdida más altos, manteniendo la pérdida más elevada entre todos los modelos. En este caso, la curva de pérdida presenta un comportamiento casi lineal decreciente.
- Es difícil atribuir los cambios observados a un factor específico, por lo que se determinarán los mejores parámetros para futuros modelos con el fin de evaluar su impacto en la serie de tiempo.

### (3) Tuneo de Hiperparámetros

En esta sección, se emplearán las utilidades del paquete `keras_tuner` para realizar una búsqueda de los mejores parámetros dentro de un rango de opciones para cada modelo de las series de tiempo. Se utilizará el proceso de `GridSearch` con el objetivo de minimizar la pérdida en el conjunto de validación para cada serie. En este caso, la búsqueda se realizará sobre los parámetros:

- **Step:** [1, 5]
- **Units** (Unidades en la capa LSTM): [1, 64], aumentando 4 unidades en cada iteración.
- **Activation:** (LSTM): ['sigmoid', 'relu', 'tanh']
- **Optimizer:** ['adam' (Adaptivo), 'sgd' (Descenso gradiente)]

```
In [ ]: def build_model(hp):
        step = hp.Int('step', min_value=1, max_value=5, step=1)
        features = 1
        units = hp.Int('units', min_value=1, max_value=64, step=4)

        inputs = keras.layers.Input((step, features))
        lstm_out = keras.layers.LSTM(units=units, activation=hp.Choice('activation', ['sigmoid', 'relu', 'tanh']))
        outputs = keras.layers.Dense(units=1)(lstm_out)

        model = keras.Model(inputs=inputs, outputs=outputs)

        model.compile(
            loss='mean_squared_error',
            optimizer=hp.Choice('optimizer', values=['adam', 'sgd'])
        )

        return model
```

```
In [ ]:
```

```
In [ ]: # Initialize the tuner
tuner = kt.GridSearch(
    build_model,
    objective='val_loss',
    max_trials=10,
    executions_per_trial=3,
    directory='tuner_outputs',
    project_name='fuel_gt_analysis'
)

# Print a summary of the search space
tuner.search_space_summary()
```

Reloading Tuner from tuner\_outputs/fuel\_gt\_analysis/tuner0.json

Search space summary

Default search space size: 4

step (Int)

```
{'default': None, 'conditions': [], 'min_value': 1, 'max_value': 5, 'step': 1, 'sampling': 'linear'}
```

units (Int)

```
{'default': None, 'conditions': [], 'min_value': 1, 'max_value': 64, 'step': 4, 'sampling': 'linear'}
```

activation (Choice)

```
{'default': 'sigmoid', 'conditions': [], 'values': ['sigmoid', 'relu', 'tanh'], 'ordered': False}
```

optimizer (Choice)

```
{'default': 'adam', 'conditions': [], 'values': ['adam', 'sgd'], 'ordered': False}
```

```
In [ ]: def tune_and_fit_model(dataset_dict, epochs):
    # Start the search
    tuner.search(
        x=dataset_dict["x_train"],
        y=dataset_dict["y_train"],
        epochs=epochs,
        validation_data=(dataset_dict["x_val"], dataset_dict["y_val"]),
        callbacks=[tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)]
    )

    # Get the optimal hyperparameters
    best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

    # Build the model with the optimal hyperparameters and train it
    model = tuner.hypermodel.build(best_hps)
    history = model.fit(
        x=dataset_dict["x_train"],
        y=dataset_dict["y_train"],
        epochs=epochs,
        validation_data=(dataset_dict["x_val"], dataset_dict["y_val"]),
        batch_size=1,
        verbose=0
    )

    # Print the best hyperparameters
    print(f"""
    The optimal number of units in the LSTM layer is {best_hps.get('units')},
    the optimal number of steps is {best_hps.get('step')},
    the optimal LSTM activation function is {best_hps.get('activation')},
    the optimal optimizer is {best_hps.get('optimizer')}.
    """)

    print("Training Loss")
    model.evaluate(
        x = dataset_dict["x_train"],
        y = dataset_dict["y_train"]
    )
    print("\nValidation Loss")
    model.evaluate(
        x = dataset_dict["x_val"],
        y = dataset_dict["y_val"]
    )
    print("\nTesting Loss")
    model.evaluate(
        x = dataset_dict["x_test"],
        y = dataset_dict["y_test"]
    )

    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='center')

    return model
```



```
In [ ]: consumption_model = tune_and_fit_model(consumption_dataset, 50)
```

The optimal number of units in the LSTM layer is 5,  
the optimal number of steps is 1,  
the optimal LSTM activation function is relu,  
the optimal optimizer is adam.

Training Loss

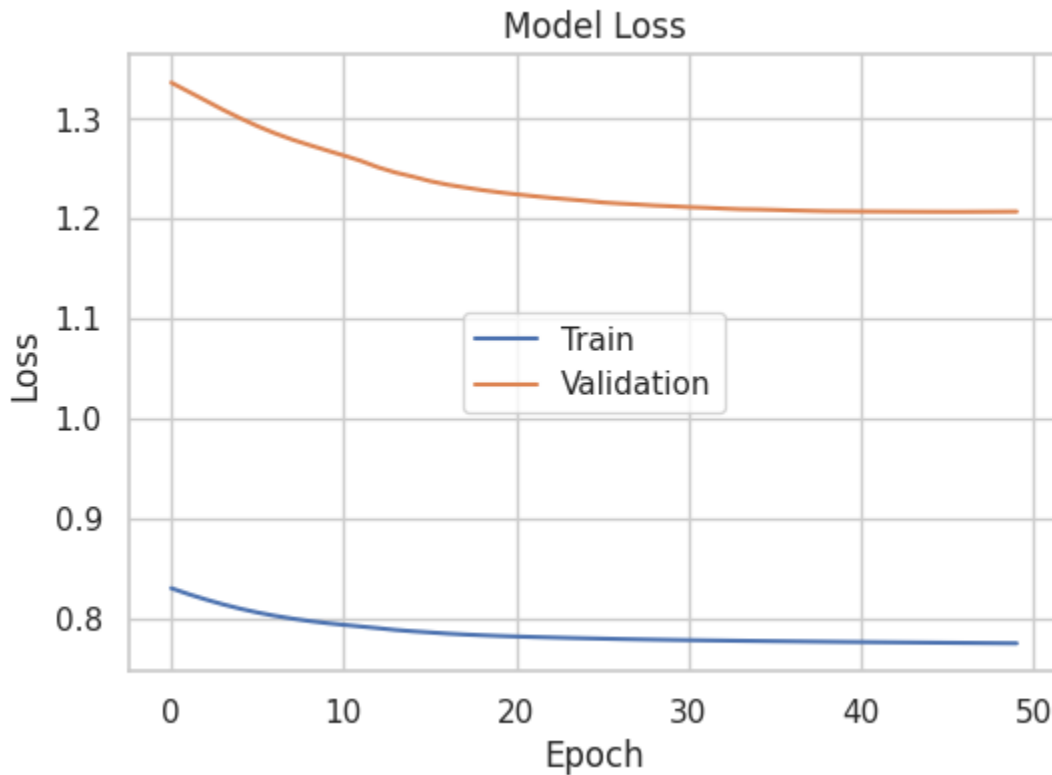
8/8 ————— 0s 2ms/step - loss: 0.6213

Validation Loss

1/1 ————— 0s 23ms/step - loss: 1.2074

Testing Loss

1/1 ————— 0s 26ms/step - loss: 2.6247



```
In [ ]: importation_model = tune_and_fit_model(importation_dataset, 50)
```

The optimal number of units in the LSTM layer is 5,  
the optimal number of steps is 1,  
the optimal LSTM activation function is relu,  
the optimal optimizer is adam.

Training Loss

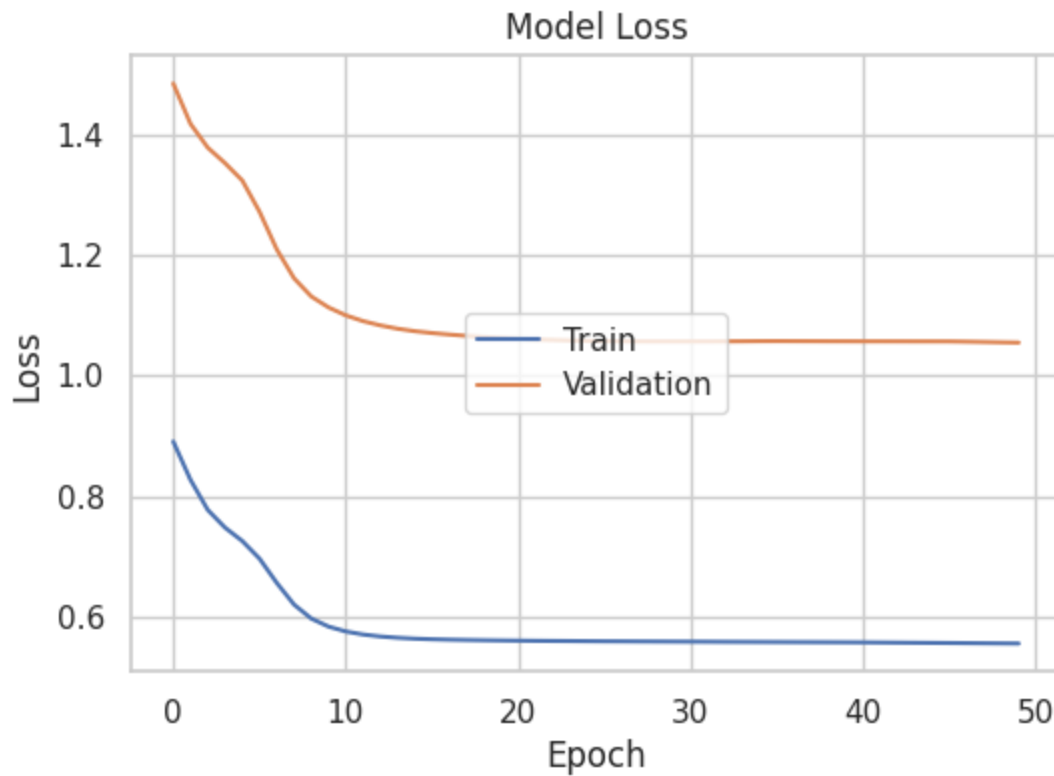
8/8 ————— 0s 3ms/step - loss: 0.3830

Validation Loss

1/1 ————— 0s 47ms/step - loss: 1.0549

Testing Loss

1/1 ————— 0s 41ms/step - loss: 0.5129



```
In [ ]: price_model = tune_and_fit_model(price_dataset, 50)
```

The optimal number of units in the LSTM layer is 5,  
the optimal number of steps is 1,  
the optimal LSTM activation function is relu,  
the optimal optimizer is adam.

Training Loss

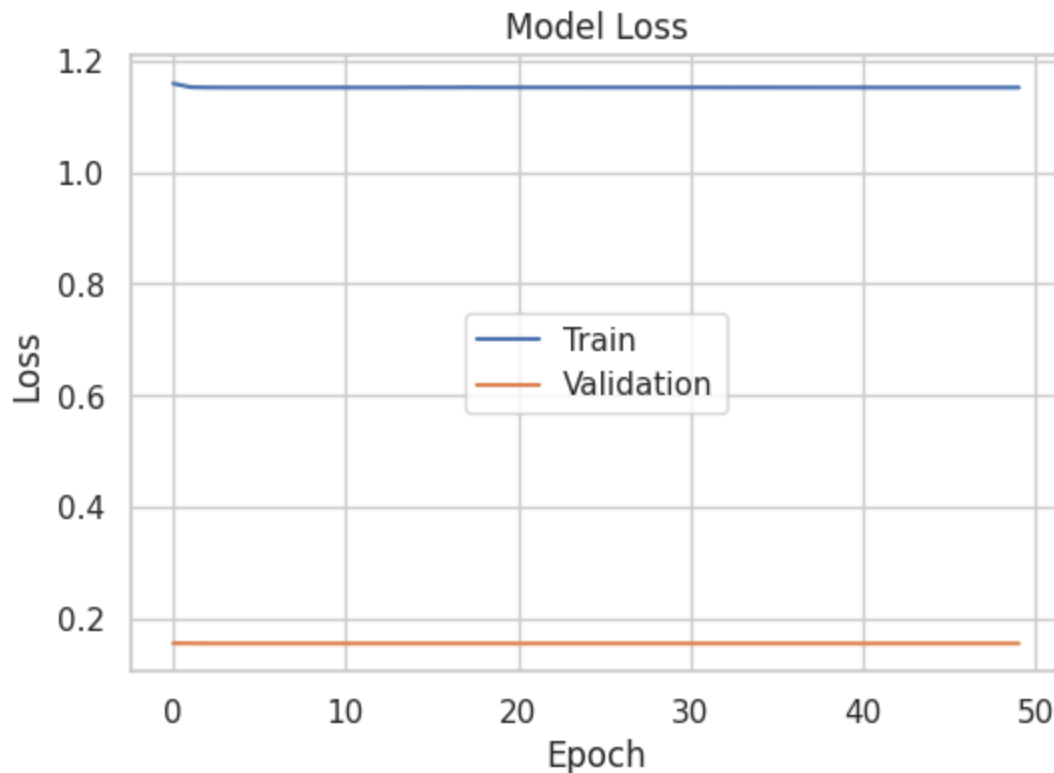
35/35 ————— 0s 1ms/step - loss: 0.8824

Validation Loss

3/3 ————— 0s 4ms/step - loss: 0.1926

Testing Loss

4/4 ————— 0s 3ms/step - loss: 0.1285



#### Observaciones 💡 -->

- A pesar de obtener valores de pérdida más bajos para todas las series de tiempo, la diferencia entre los modelos con parámetros optimizados y los modelos con parámetros arbitrarios no es significativa.
- El comportamiento de la serie de precios es idéntica a las anteriores, potencialmente exhibiendo alguna deficiencia en la preparación del conjunto de datos o mal ajuste del modelo para esta serie de tiempo.
- Las curvas de pérdida para la serie de consumo e importación es más pronunciada, acercándose más a un comportamiento deseado en el cambio de pérdida a través del tiempo.
- Los mejores parámetros para todos los modelos son los mismos, por lo que se puede inferir que el rendimiento de los modelos depende solamente de la serie de tiempo en la que se entrena.
- No hay necesidad de agregar más épocas al entrenamiento de los modelos ya que todos los modelos convergen aproximadamente a un valor final.

#### (4) Predicciones

Para cada serie de tiempo, el mejor modelo es el generado luego del ajuste de parámetros en cuanto a pérdida. Por lo tanto, para realizar predicciones, se utilizarán estos modelos.

```
In [ ]: def predict(model, scaler, dataset, series, index_series, n_steps):  
        prediction = [0] * len(dataset)
```

```

i = 0
for X in dataset:
    X = np.reshape(X, (1,1,1))
    yhat = model.predict(X, verbose=0)
    yhat = scaler.inverse_transform(yhat)
    yhat += series[i+1]
    prediction[i] = yhat[0][0]
    i += 1
prediction = pd.DataFrame(prediction, index=index_series.index)
return prediction

```

```

In [ ]: def plot_predictions(val_pred, test_pred, true_values, title, ylabel, offset
# Plot the original data
plt.plot(true_values, label='True Values', color='#55c667', linestyle='-')

# Plot the predicted data on top of the original data
plt.plot(val_pred, label='Predicted Validation Data', color='#3b528b', lin
plt.plot(test_pred, label='Predicted Test Data', color='#440154', linestyle

# Adding titles and labels
plt.title(title)
plt.xlabel('Date')
plt.ylabel(ylabel)
plt.legend()

# Show the plot
plt.show()

# Split true values into validation and test parts
true_val = true_values.iloc[:len(val_pred)]
true_test = true_values.iloc[len(val_pred):-offset]

# Calculate MAE and RMSE for validation data
mae_val = mean_absolute_error(true_val, val_pred)
rmse_val = np.sqrt(mean_squared_error(true_val, val_pred))

# Calculate MAE and RMSE for test data
mae_test = mean_absolute_error(true_test, test_pred)
rmse_test = np.sqrt(mean_squared_error(true_test, test_pred))

# Print the MSE and RMSE values
print(f"Validation Data - MAE: {mae_val:.4f}, RMSE: {rmse_val:.4f}")
print(f"Test Data - MAE: {mae_test:.4f}, RMSE: {rmse_test:.4f}")

```

## (1) Modelo de Consumo de Diesel

### Modelo LSTM

```

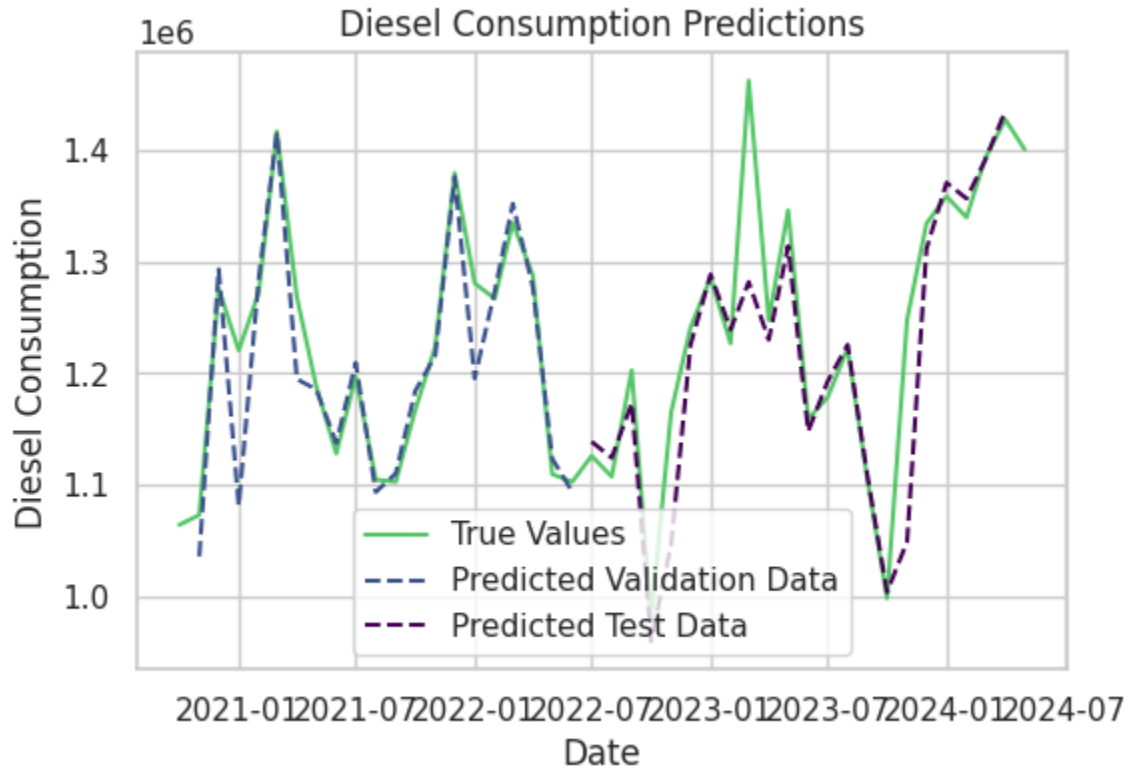
In [ ]: # Concatenate the validation and test datasets
train_consumption_original, val_consumption_original, test_consumption_origi
combined_dataset = np.concatenate((consumption_dataset["x_val"], consumption
combined_series = pd.concat([val_consumption_original, test_consumption_orig

# Run predictions on the combined dataset
combined_predictions = predict(consumption_model, consumption_scaler, combin

```

```
# Split predictions into separate predictions for x_val and x_test
n_val = len(consumption_dataset["x_val"])
val_predictions = combined_predictions[:n_val]
test_predictions = combined_predictions[n_val:]

plot_predictions(val_predictions, test_predictions, pd.concat([val_consumpti
```

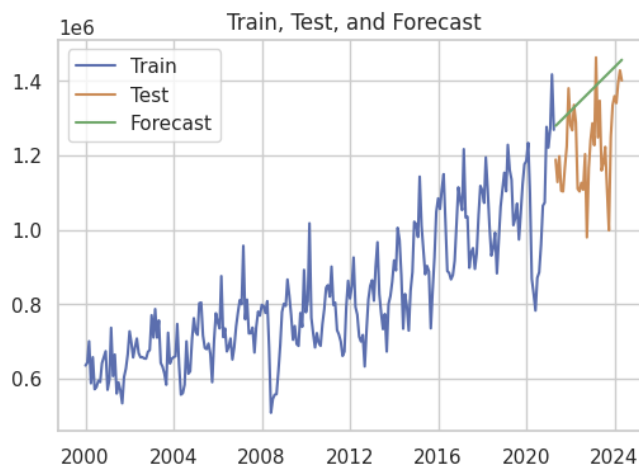


Validation Data - MAE: 99908.4753, RMSE: 121099.8418

Test Data - MAE: 75745.7080, RMSE: 99507.1769

### Modelo ARIMA

```
diesel_consumption_ARIMA_pred = predict_and_plot(diesel_consumption_ARIMA, consumption_train, consumption_test)
```



Mean Absolute Error (MAE): 147982.40

Root Mean Squared Error (RMSE): 176612.48

## Observaciones 💡 -->

- Para el **modelo LSTM**, el error absoluto medio (MAE) para la validación (99908,4753) es significativamente mayor que el de los datos de prueba (75745,7080), sugiriendo así que el modelo puede generalizar mejor a datos no vistos en comparación con el conjunto de validación.
- A partir de las discrepancias entre la validación y la prueba MAE/RMSE, es posible inferir que el modelo podría estar sobreajustándose a los datos de validación.
- Por su parte, para el modelo **ARIMA** se evidencia un nivel moderado de error de predicción promedio, similar a los resultados de la validación de LSTM.
- El RMSE es bastante alto, lo que sugiere discrepancias significativas entre los valores previstos y reales.

## Comparación 🗖️ -->

### Consistencia del Rendimiento:

- El modelo LSTM muestra mejor generalización a datos no vistos, evidenciada por métricas de error en test más bajas que las métricas de validación y test de ARIMA, lo que sugiere que el modelo LSTM es más robusto en general.

### Tipo de Modelo y Complejidad:

- El modelo LSTM es una arquitectura de aprendizaje profundo más compleja, mientras que ARIMA es un método tradicional de pronóstico de series temporales. El RMSE más alto de ARIMA puede indicar que le cuesta capturar la complejidad de los patrones subyacentes en los datos de consumo de diésel en comparación con LSTM.

**Nota:** Basándose en lo anteriormente expuesto, se concluye que el modelo más adecuado para la predicción de la serie de tiempo relacionada con el **consumo de diésel** en Guatemala es el modelo LSTM.

## (2) Regular Gasoline Importation

### Modelo LSTM

```
In [ ]: train_importation_original, val_importation_original, test_importation_origi
combined_dataset = np.concatenate((importation_dataset["x_val"], importation
combined_series = pd.concat([val_importation_original, test_importation_orig

combined_predictions = predict(importation_model, importation_scaler, combin

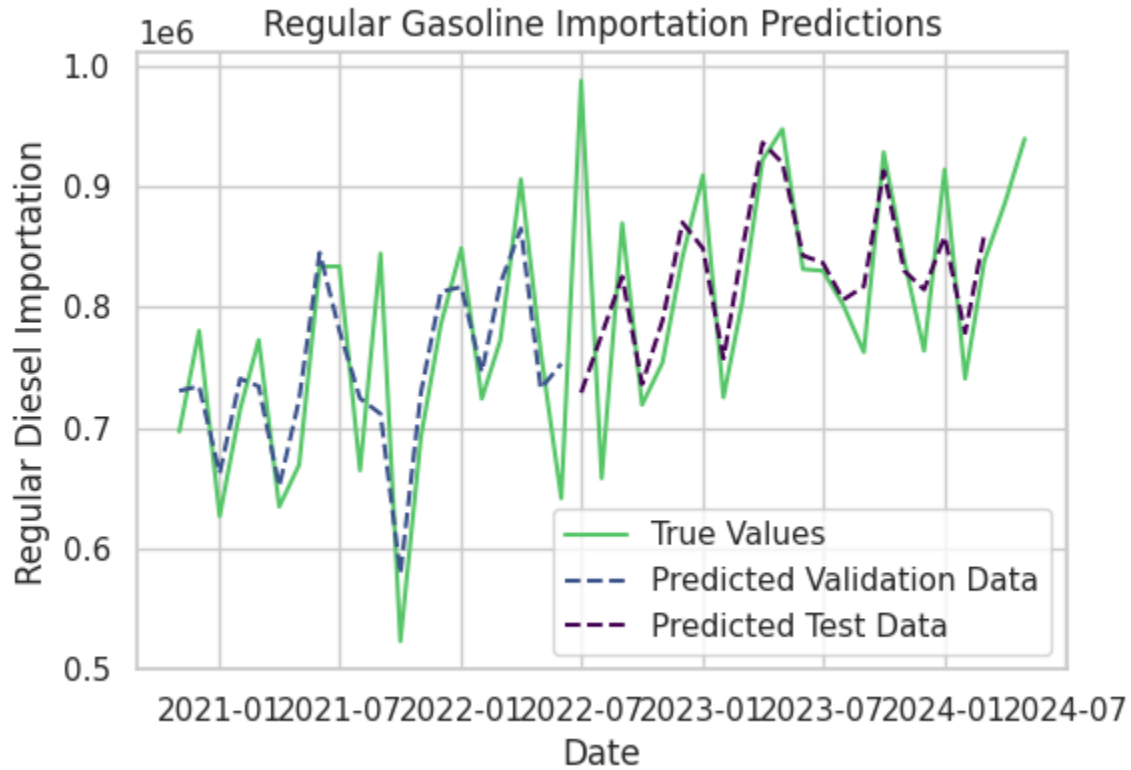
# Split predictions into separate predictions for x_val and x_test
```

```

n_val = len(importation_dataset["x_val"])
val_predictions = combined_predictions[:n_val]
test_predictions = combined_predictions[n_val:]

plot_predictions(val_predictions, test_predictions, pd.concat([val_importati

```

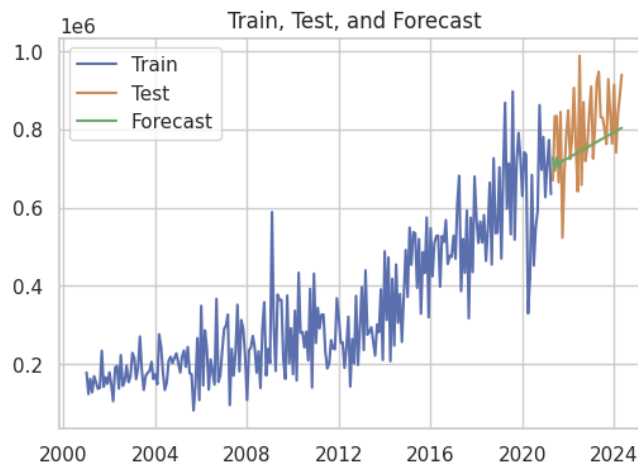


Validation Data - MAE: 45612.7146, RMSE: 53919.1782

Test Data - MAE: 96786.8933, RMSE: 116913.3359

### Modelo ARIMA

```
regular_importation_ARIMA_pred = predict_and_plot(regular_importation_ARIMA, importation_train, importation_test)
```



Mean Absolute Error (MAE): 85512.88  
Root Mean Squared Error (RMSE): 103823.00

Observaciones 💡 -->

- Para el *modelo LSTM*, el error absoluto medio (MAE) para la validación (45612.7146) es significativamente mayor que el de los datos de prueba (96786.8933), nuevamente sugiriendo así que el modelo puede generalizar mejor a datos no vistos en comparación con el conjunto de validación.
- En general, parece ser que el modelo tiene más dificultades con errores mayores en el conjunto de pruebas, lo que refleja un posible sobreajuste o diferencias en la distribución de datos.
- Por su parte, los resultados del modelo **ARIMA** indican un mejor rendimiento de predicción promedio en los datos de prueba en contraste con el modelo LSTM.
- El RMSE también es más bajo que el del modelo LSTM, lo que sugiere que el modelo ARIMA tiene menos errores de predicción y menos severos en el conjunto de prueba en comparación con el LSTM.

### Comparación 🗉 -->

#### Magnitudes de Error:

- Las métricas de test del modelo ARIMA demuestran un mejor rendimiento, con MAE y RMSE más bajos, lo que indica que generaliza mejor a los datos de test.

#### Generalización:

- La discrepancia en el rendimiento de test sugiere que el modelo LSTM puede estar sobreajustando los datos de validación y teniendo dificultades para adaptarse a los patrones diferentes en los datos de test.

Sin embargo, algo **muy** importante a tomar en consideración es el hecho de que el modelo ARIMA presenta una predicción de forma similar a la de una función lineal. Esto parece causar la falta de captura de las fluctuaciones o la estacionalidad en la serie de tiempo, lo que podría generar un ajuste insuficiente si los datos reales muestran patrones más complejos. Pr ende, aunque sus métricas sean mejores, se cree no es lo suficientemente complejo como para capturar los patrones reales en los datos.

**Nota:** Basándose en lo anteriormente expuesto, se concluye que el modelo más adecuado para la predicción de la serie de tiempo relacionada con el **importación de gasolina regular** en Guatemala es el modelo LSTM.

### (3) Precio de Diesel

#### Modelo LSTM

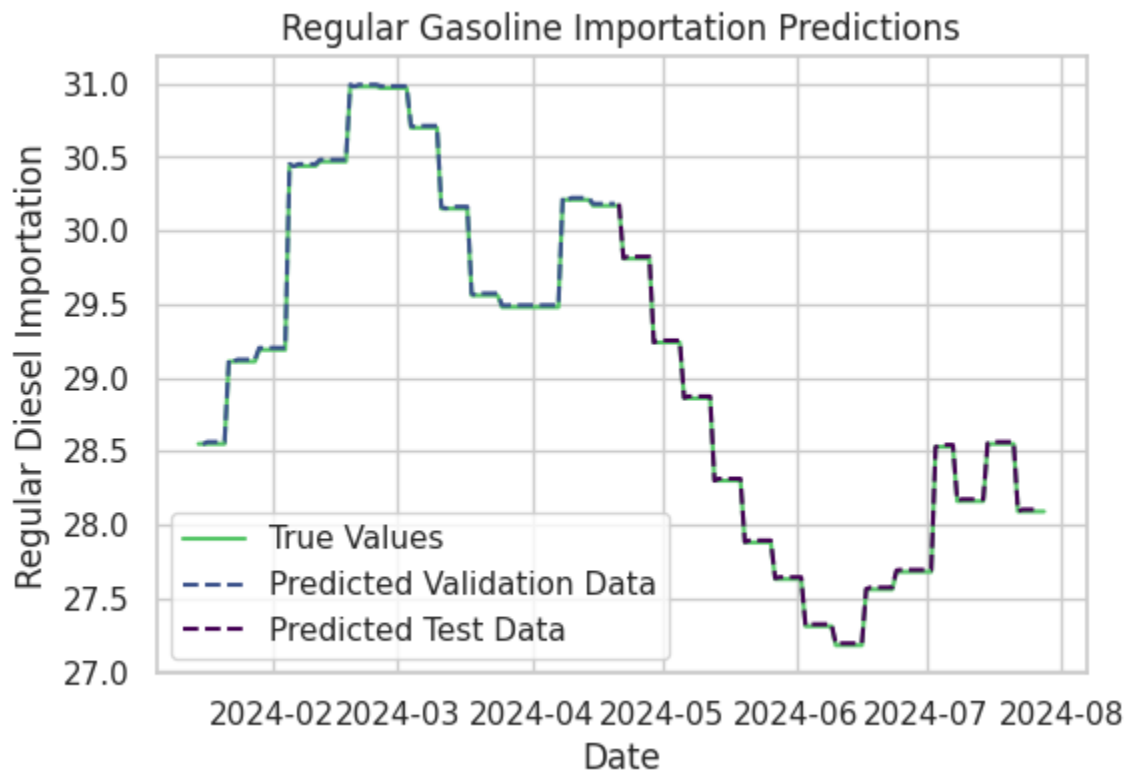


```
In [ ]: train_price_original, val_price_original, test_price_original = train_val_te
combined_dataset = np.concatenate((price_dataset["x_val"], price_dataset["x_
combined_series = pd.concat([val_price_original, test_price_original], axis=

combined_predictions = predict(price_model, price_scaler, combined_dataset,

# Split predictions into separate predictions for x_val and x_test
n_val = len(price_dataset["x_val"])
val_predictions = combined_predictions[:n_val]
test_predictions = combined_predictions[n_val:]

plot_predictions(val_predictions, test_predictions, pd.concat([val_price_ori
```

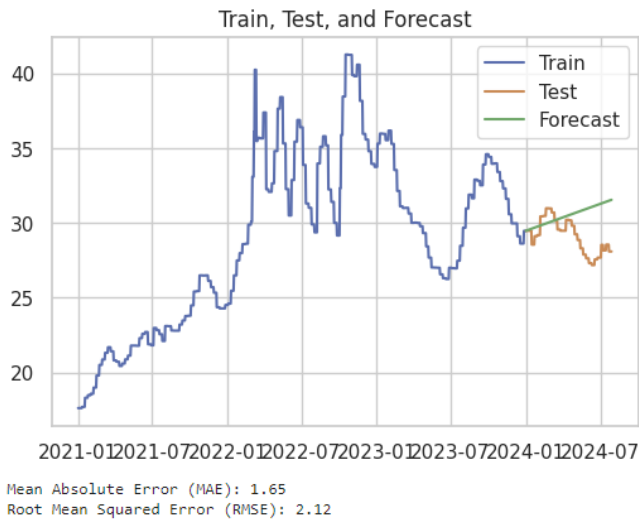


Validation Data - MAE: 0.0599, RMSE: 0.1901

Test Data - MAE: 0.0682, RMSE: 0.1645

#### Modelo ARIMA

```
diesel_price_ARIMA_pred = predict_and_plot(diesel_price_ARIMA, price_train, price_test)
```



### Observaciones 💡 -->

- El modelo **LSTM** mostró un excelente rendimiento en los datos de validación y prueba. Con un MAE de 0.0599 en el conjunto de validación, indica un bajo error promedio de predicción, lo que sugiere que el modelo captura eficazmente los patrones subyacentes.
- Además, el RMSE de 0.1645 en los datos de prueba es notablemente bajo, lo que refleja la capacidad del modelo para minimizar errores en predicciones sobre datos no vistos, destacando su capacidad de generalización.
- El modelo **ARIMA**, aunque útil, presentó un desempeño inferior en comparación con el modelo LSTM. Su MAE en los datos de prueba fue de 1.65, lo que sugiere un mayor error promedio de predicción, indicando que el modelo tiene dificultades para capturar patrones en los datos. Asimismo, el RMSE de 2.12 en los datos de prueba es significativamente más alto que el del LSTM, lo que indica que los errores de predicción son más graves, afectando su eficacia general en comparación con el modelo LSTM.

### Comparación 🏆 -->

#### Magnitudes de Error:

- Ambos modelos se desempeñan bien en general, pero el modelo LSTM demuestra una precisión superior en términos de MAE y RMSE en los conjuntos de datos de validación y test.
- Las métricas de error bajas del LSTM sugieren que captura tendencias y fluctuaciones en los datos de manera más efectiva que el modelo ARIMA.

**Generalización:**

- El modelo LSTM muestra una excelente generalización, como lo demuestra su rendimiento consistente en los datos de validación y test. El aumento en el MAE de validación a test es mínimo, lo que indica robustez.
- Por su parte, el modelo ARIMA, aunque se desempeña razonablemente bien, muestra métricas de error más altas, lo que puede reflejar limitaciones en la captura de patrones más complejos en los datos.

**Nota:** Basándose en lo anteriormente expuesto, se concluye que el modelo más adecuado para la predicción de la serie de tiempo relacionada con el **costo de Diesel** en Guatemala es el modelo LSTM.