UNIVERSITY
OF BRESCIA

DEPARTMENT OF INFORMATION ENGINEERING

Master's Degree

in Computer Engineering

Internship Report

Time series prediction: a study of the combination of LSTM with frequency analysis

Teacher: Prof. Emilio Alfonso Gerevini

Internship tutor: Dr. Luca Piccinelli

Student:

Giacomo Golino (719210)

Contents

Introduction

This report aims to explore potential improvements in time series prediction by integrating frequency analysis into a Long Short-Term Memory (LSTM) neural network. Time series data, inherently sequential and commonly found in domains such as finance, meteorology, and signal processing, often contains periodic or cyclical patterns that traditional LSTM networks may overlook, despite their effectiveness in capturing long-term dependencies. The primary focus of this study is to investigate whether applying a frequency-based analysis to the input data before feeding it into the LSTM model can enhance its predictive capabilities. The hypothesis is that combining the strength of LSTMs in capturing temporal dependencies with insights from frequency components could yield higher accuracy, especially in datasets where periodicity plays a significant role.

To test this hypothesis, this work examines various model architectures LSTM and BiLSTM variants with Fourier-transformed components. A core part of the approach involves focusing on common signal patterns within the frequency domain, employing attention mechanisms and dynamic frequency selection to capture recurring characteristics across signals. The methodology begins by generating synthetic datasets that include trends, noise, and periodic elements, allowing for controlled experimentation with complex signal patterns.

This work will delve into the underlying theory of time series analysis, examine LSTM network mechanisms, and propose an approach that leverages frequency-domain information to complement standard time-domain processing. Through empirical experiments and comparisons with baseline models, we aim to assess whether integrating frequency analysis leads to improved performance and under what conditions these improvements are most effective. The results of this research are intended to reveal whether the proposed method offers a more effective approach for forecasting trends in time series data, particularly when recurring frequency-driven patterns are present.

1. Preparation

1.1 Data Preparation

For this case study, I opted not to use pre-existing datasets. Instead, I chose to generate the data in a simpler, more controlled manner. While I developed several methods for data generation, I will only present the most relevant ones here. The full code is available here: *"Colab Notebook"*.

Some methods are adapted from the article *"A Step-by-Step Walkthrough Neural Networks for Time-series Forecasting"* by Luca Piccinelli.

1.1.1 Wave Generation Method

Below is the Python code for a method that generates a wave based on a specified frequency, resolution, and amplitude. This function can use different mathematical functions, with the sine function as the default.

```
import numpy as np


def wave(frequency: int, resolution: int, amplitude: int, fn=np.sin):
    length = np.pi * 2 * frequency
    my_wave = fn(np.arange(0, length, length / resolution))
    return my_wave * amplitude
```

1.1.2 Trend Method

The following Python function generates a trend based on a logistic curve. It accepts parameters for the curve's steepness ('c1'), shift ('c2'), and amplitude, and returns a trend signal over the specified length.

```
def trend(length, c1=1.0, c2=0.0, amplitude=1.0):
    xs = np.linspace(0, 1, length)
    ys = 1.0 / (1 + np.exp(-c1 * (xs - c2)))
    return ys * amplitude
```

1.1.3 Noise Method

This function generates noise based on a normal distribution. The "std" parameter controls the standard deviation of the noise, and the amplitude can be adjusted to scale the noise.

```
import numpy as np

def noise(resolution, std, amplitude=1):
    return np.random.normal(0, std, resolution) * amplitude
```

1.2 Input Generation

Finally, we can generate our initial dataset by using the previously defined methods. We will create a synthetic time series by combining different waveforms (sine and cosine waves), noise, and trends. This synthetic data will simulate a time series with various components, which will later be used for training our model.

The following Python code generates the input data:

```
# Define the total number of points
res = 520

# Generate input data using previously defined methods
sine_wave = wave(10, res, 1) + wave(20, res, 3) + wave(5, res, 2, np.cos)
    + noise(res, 3) + trend(res)
```

```
# Plot the generated data
plt.figure(figsize=(12, 4))
plt.plot(sine_wave)
plt.title('Sales Over Time')
plt.xlabel('Time')
plt.ylabel('Value')
plt.show()
```

The output of this code will produce an wave such like this one:

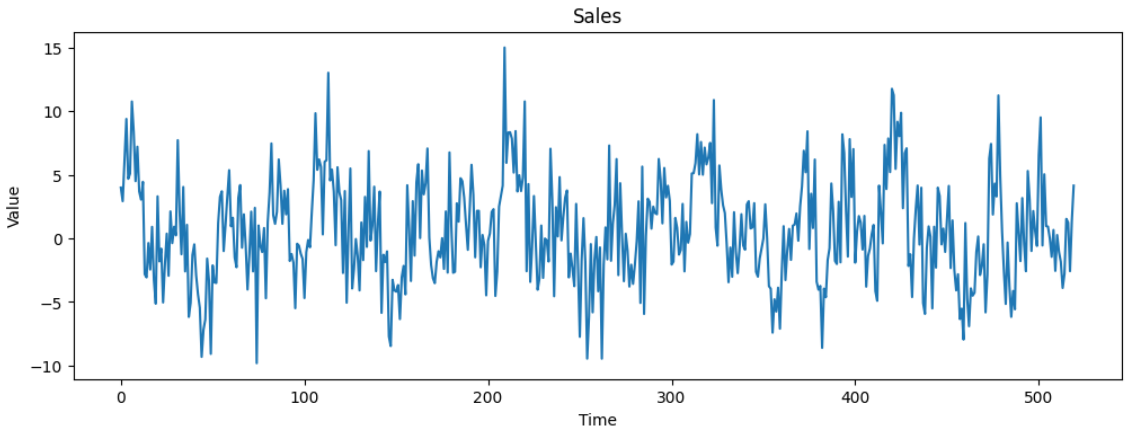


Figure 1.1: Initial data

1.3 Dataset Generation

An essential step in solving this problem is the creation of a dataset with meaningful values, but how do we achieve this? The main idea is to generate lagged data. By "lagged data," we mean taking a subsequence of values from the original time series (i.e., a subset of days) and linking them to the next day. This method allows our LSTM to analyze and, hopefully, detect time series correlations within the data.

1.3.1 Dataset Creation Method

The following Python function creates a dataset by taking a sequence of values from the original series and preparing them for supervised learning, where the input data

is the past sequence, and the target is the next value in the series.

```
import numpy as np


def create_dataset(series, sequence_length):
    forecast_horizon = 1
    X = []
    y = []
    for i in range(len(series) - sequence_length):
        X.append(series[i:(i + sequence_length)])
        y.append(series[i + sequence_length + forecast_horizon - 1])
    X = np.array(X)
    y = np.array(y)
    return X, y
```

For example, consider a time series of consecutive numbers:

Series:

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$

Using a sequence length of 3, the function generates the following input and output datasets:

```
series = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
sequence_length = 3


X, y = create_dataset(series, sequence_length)
```

```
# Outputs
print("X:", X)
print("y:", y)
print("X shape:", X.shape)
print("y shape:", y.shape)
```

Output:

X (input sequences):

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \\ 5 & 6 & 7 \\ 6 & 7 & 8 \end{bmatrix}$$

y (next value for each sequence):

$$[3, 4, 5, 6, 7, 8, 9]$$

Shapes:

- **X shape:** (7, 3)
- **y shape:** (7,)

In this example, the function creates 7 input sequences of length 3 and their corresponding target values (the next value in the sequence). This structure allows the LSTM to learn from past data to predict future values.

1.4 Train, Validation, Test Set Generation

In order to thoroughly evaluate the model's performance, it is essential to split the dataset into three sets: training, validation, and testing. The training set is used to train the model, the validation set is used to fine-tune and optimize hyperparameters, and the test set evaluates the model's ability to generalize on unseen data. Below is a Python function that splits the dataset into these three parts based on specified ratios.

1.4.1 Dataset Division Method

The following Python function divides the dataset into training, validation, and test sets. The training ratio specifies the portion of the dataset used for training, while the validation ratio defines the split between the validation and test sets.

```
def divide_dataset_validation(X, y, train_ratio=0.7,
    validation_ratio=0.5):
    train_size = int(len(X) * train_ratio)
    validation_size = int((len(X) - train_size) * validation_ratio)

    X_train, X_validation = X[:train_size], X[train_size:train_size +
        validation_size]
    y_train, y_validation = y[:train_size], y[train_size:train_size +
        validation_size]
    X_test, y_test = X[train_size + validation_size:], y[train_size +
        validation_size:]
    return X_train, y_train, X_validation, y_validation, X_test, y_test

# Using previously defined variables (X and y)
X_train, y_train, X_validation, y_validation, X_test, y_test =
    divide_dataset_validation(X, y, validation_ratio=0.4)
```

In this example, the dataset is divided into 70% for training, and the remaining 30% is split between validation and testing based on the "`validation_ratio`". With a `validation_ratio` of 0.4, 40% of the remaining data will be used for validation, and 60% will be for testing.

Example Output:

Given the previously defined "X" and "y", the dataset is divided as follows:

X_train (Training Set):

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

y_train:

$$\begin{bmatrix} 3 \\ 4 \\ 5 \\ 6 \end{bmatrix}$$

X_validation (Validation Set):

$$\begin{bmatrix} 4 & 5 & 6 \end{bmatrix}$$

y_validation:

$$[7]$$

X_test (Test Set):

$$\begin{bmatrix} 5 & 6 & 7 \\ 6 & 7 & 8 \end{bmatrix}$$

y_test:

$$\begin{bmatrix} 8 \\ 9 \end{bmatrix}$$

This method ensures that the dataset is split appropriately to allow the model to train, validate, and test on different parts of the data. By using these ratios, the dataset is divided into meaningful portions for model evaluation.

2. Long Short Term Memory

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) that are well-suited for modeling sequential data, such as time series, natural language, and other types of data where order and dependencies across time are important. Unlike traditional RNNs, which suffer from the vanishing gradient problem, LSTMs are designed to capture long-range dependencies by using a series of gates to regulate the flow of information.

An LSTM cell consists of three main gates:

- **Forget Gate**: Determines what information from the previous time step should be discarded.
- **Input Gate**: Controls what new information is added to the cell state.
- **Output Gate**: Dictates how much of the cell state should be passed to the next time step.

These gates allow LSTMs to retain information over long sequences, making them particularly powerful for tasks that require understanding of both short-term and long-term patterns.

2.1 LSTM Architecture

LSTMs maintain an internal memory state that is updated through these gates as the network processes the sequence step by step. The internal cell state helps the LSTM remember important information from earlier in the sequence, while irrelevant information is forgotten.

Below is a simplified diagram of an LSTM cell, showing how the gates interact with the cell state and the hidden state.

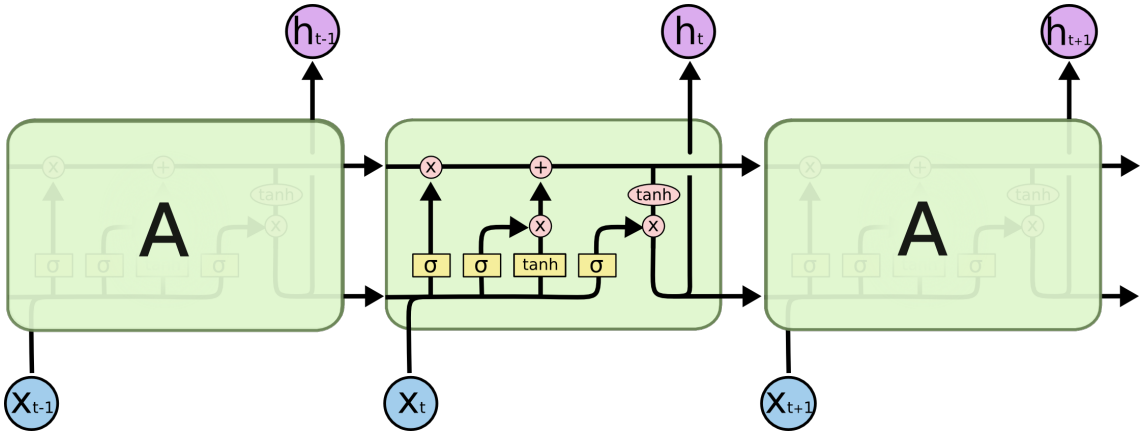


Figure 2.1: LSTM Cell Structure

Figure taken from: Colah's Blog on Understanding LSTMs
(`https://colah.github.io/posts/2015-08-Understanding-LSTMs/`)

This diagram illustrates the key components of an LSTM unit: the cell state, the hidden state, and the three gates (forget, input, and output). By controlling the flow of information through these gates, LSTMs are able to mitigate issues like short-term memory loss, which are common in traditional RNNs.

2.2 LSTM Model Implementation

In this section, I describe the implementation of a simple Long Short-Term Memory (LSTM) network used for time series prediction. The primary objective is to capture temporal dependencies in the data, and LSTMs are well-suited for this purpose due to their ability to maintain long-term memory of past data points.

2.2.1 Scaling of Input and Output Data

Before feeding the data into the LSTM network, it is crucial to scale both the input and output data. This step ensures that all features are within a similar range, preventing the model from being biased towards features with larger magnitudes. Since LSTM networks are sensitive to the scale of input data, failing to scale the data could lead to slower convergence or suboptimal performance.

In this implementation, I used the `MinMaxScaler` from `sklearn`, which scales the data to the range [0, 1]. The dataset was reshaped before scaling to ensure compatibility with the scaler, and then it was transformed back to its original shape after scaling. Both the input features (X) and the target values (y) were scaled:

```
# Scaling the input data (X)
scaler_X = MinMaxScaler(feature_range=(0, 1))

# Reshaping for scaling
X_train_reshaped = X_train.reshape(-1, 1)
X_validation_reshaped = X_validation.reshape(-1, 1)
X_test_reshaped = X_test.reshape(-1, 1)

# Fitting the scaler and applying it to training, validation, and test
    sets
scaler_X.fit(X_train_reshaped)

X_train_scaled =
    scaler_X.transform(X_train_reshaped).reshape(X_train.shape)

X_validation_scaled =
    scaler_X.transform(X_validation_reshaped).reshape(X_validation.shape)

X_test_scaled = scaler_X.transform(X_test_reshaped).reshape(X_test.shape)

# Scaling the target (y)
scaler_y = MinMaxScaler(feature_range=(0, 1))
y_train_scaled = scaler_y.transform(y_train.reshape(-1, 1)).flatten()
y_validation_scaled = scaler_y.transform(y_validation.reshape(-1,
    1)).flatten()
y_test_scaled = scaler_y.transform(y_test.reshape(-1, 1)).flatten()
```

This scaling process ensures that the input features and target values are within a consistent range, improving model training and performance. Moreover, by dividing the dataset into training, validation, and test sets, we prevent data leakage, which is crucial to avoid overly optimistic performance results. Keeping the test data separate ensures that the model is evaluated on unseen data, providing a more realistic measure of its generalization ability. Furthermore, by scaling the data before feeding it to the model and inverting the scaling after making predictions, we guarantee that the error metrics, such as the mean squared error (MSE), are computed on the same scale as the original data, which allows for meaningful interpretation of the results.

The formula to calculate **mean squared error** (MSE) is:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.1}$$

where:

- n is the number of data points,
- y_i is the actual value,
- $\hat{y}_i$ is the predicted value.

2.2.2 LSTM Network Architecture

The LSTM network is composed of a sequence of layers designed to process time series data. The architecture is relatively simple but effective for this task. Below is a breakdown of the model structure:

- **Input Layer**: The input to the model is a sequence of length `sequence_length`, reshaped into a 3D format that LSTMs require.
- **LSTM Layer**: The LSTM layer contains 128 units with a ReLU activation function. It captures the temporal dependencies in the input time series and outputs a fixed-length representation of the sequence.
- **Dense Layers**: After the LSTM layer, two Dense layers are used. The first Dense layer has 64 units and a ReLU activation function to capture higher-level representations of the sequence. The final Dense layer has a single unit

to produce the predicted value for the next time step.

Here is the implementation of the LSTM model:

```
# Input layer
input_sales = Input(shape=(sequence_length, 1))

# LSTM layer with 128 units and ReLU activation
x_time = LSTM(128, activation='relu', return_sequences=False)(input_sales)

# Dense layers for the output
x_output = Dense(64, activation='relu')(x_time)
x_output = Dense(1)(x_output)

# Model definition
model = Model(inputs=input_sales, outputs=x_output)

# Compile the model with RMSprop optimizer and mean squared error loss
model.compile(optimizer='rmsprop', loss='mse')

# Model summary
model.summary()
```

The last line of the previous code permits to obtain a visual representation of the written model. Our LSTM model has the following structure:

Layer (type)	Output Shape	Param #
input_layer_24 (InputLayer)	(None, 30, 1)	0
lstm_24 (LSTM)	(None, 128)	66,560
dense_95 (Dense)	(None, 64)	8,256
dense_96 (Dense)	(None, 1)	65

Figure 2.2: Summarization of LSTM structure

2.2.3 Model Training and Results

To prevent overfitting, early stopping was employed during training. The model was trained for a maximum of 200 epochs, but training stopped early if the validation loss did not improve for 10 consecutive epochs.

```
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

# Train the model with validation data
history = model.fit(
    X_train_scaled, y_train_scaled,
    epochs=200,
    batch_size=64,
    validation_data=(X_validation_scaled, y_validation_scaled),
    callbacks=[early_stopping],
    verbose=1
)
```

After training, the model was evaluated on the test set, and the results were plotted to show the comparison between real and predicted values.

```
# Predict on test data
y_pred_scaled = model.predict(X_test_scaled)

# Inverse transform the predictions
y_pred = scaler_y.inverse_transform(y_pred_scaled)

# Inverse transform the actual test targets
y_test_inv = scaler_y.inverse_transform(y_test_scaled.reshape(-1, 1))

# Plot the real vs predicted values
plt.figure(figsize=(12, 6))
plt.plot(y_test_inv, label='Real Values', marker='o')
plt.plot(y_pred, label='Predicted Values', marker='o')
plt.title('Real vs Predicted Values')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```

The resulting plot is:

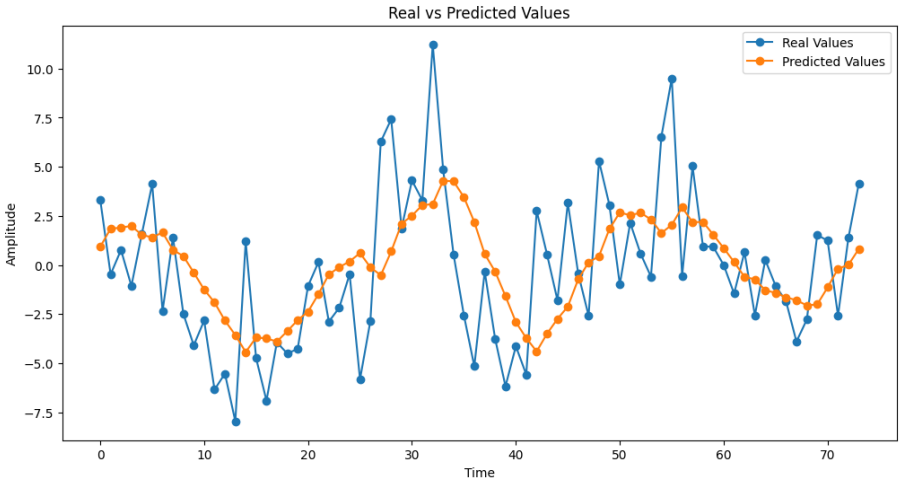


Figure 2.3: LSTM prediction value compared to the real ones.

with an ***MSE*** (2.2.1) equal to: 11.458

2.2.4 Why this LSTM

The Long Short-Term Memory model presented here is designed and modeled to serve as a baseline model. This baseline will be used as a reference point for comparison with more advanced models that will be developed later in our work. The primary goal of this LSTM is to establish a foundation for evaluating the improvements achieved with the subsequent models. By using a simple and standard architecture, such as the LSTM, we ensure that any performance gains from our custom models are measured against a reliable and commonly used model in time series forecasting tasks.

3. LSTM and Fourier Transformed Models

After implementing a baseline LSTM model, I developed different models to address specific limitations and improve performance. These models incorporate variations in the architecture, including Fourier-transformed layers, attention mechanisms, and activation functions. Each model is designed with a unique approach to enhance prediction accuracy for time series data by either capturing more complex patterns or adjusting how attention is distributed across frequency components.

The primary goal of these models is to compare their performance against the baseline LSTM and explore how different modifications contribute to handling time series prediction tasks more effectively. The models can also be useful for diverse tasks where data exhibits periodicity, seasonality, or underlying hidden patterns in both time and frequency domains.

Before showing the model, it is essential to understand how the Softmax layer operates. The Softmax function is commonly used in neural networks to normalize the outputs of a layer, converting them into probabilities. It ensures that the sum of all outputs equals 1, effectively creating a probability distribution. This is particularly useful in attention mechanisms, where the model needs to focus on the most relevant components of the input data.

The formula for Softmax is:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{3.1}$$

Where:

- x_i: is the i^{th} input element.
- e^{x_i} is the exponential of the input element.

It's important to enlight that the denominator is the sum of all exponentials of the input elements, ensuring the outputs sum to 1 (each output will have a value $\epsilon[0, 1]$).

In this model, Softmax is applied to the frequency domain data. After transforming the input data with a Fourier transform, the Softmax layer is used to calculate attention scores for each frequency component. The higher the Softmax score for a frequency, the more the model focuses on that frequency during prediction. This enables the model to prioritize important periodic components in the time series data.

3.1 Version 0

In this paper I'll enumerate the model version starting from number 0. At the beginning of each version section I'll provide an explanation of what are the differences between the current model and the previous one.

In order to have a shorter explanation I'll explain the differences between the first "subversion" of each model version.

3.1.1 Fourier Transformed LSTM with Softmax Attention

The first model builds on the LSTM architecture but incorporates Fourier-transformed inputs and a softmax attention mechanism. The motivation for this model is to capture both time-domain dependencies using the LSTM and frequency-domain information using the Fourier transform. The softmax attention layer helps the model to focus on the most relevant frequency components, giving more importance to patterns that significantly contribute to the prediction task.

Model Structure

- **LSTM Layer**: extracts temporal dependencies from the input sequence.
- **Fourier Transform Layer**: transforms the input sequence into the frequency domain, capturing periodic patterns.
- **Attention Layer**: Softmax attention helps focus on important frequency components.
- **Dense layer**: after processing both time and frequency data, dense layers combine the features for final predictions.

This model is useful for time series that contain periodic components, such as sales data with weekly or yearly seasonality.

```
# Fourier Transformed LSTM with Softmax Attention
input_sales = Input(shape=(sequence_length, 1))

# LSTM Layer for time-domain analysis
x_time = LSTM(64, return_sequences=False)(input_sales)

# Fourier Transform Layer
x_freq = Lambda(fourier_transform_layer)(input_sales)

# Softmax Attention
attention_scores = Dense(fft_length, activation='tanh')(x_freq)
attention_weights = Softmax()(attention_scores)
x_freq_attended = Multiply()([x_freq, attention_weights])

# Dense Layer for combined analysis
x_combined = Concatenate()([x_time, x_freq_attended])
x_output = Dense(64, activation='relu')(x_combined)
x_output = Dense(1)(x_output) # Future sales prediction
model = Model(inputs=input_sales, outputs=x_output)
```

The result of this code is a model with the following structure:

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 30, 1)	0	-
lambda (Lambda)	(None, 16)	0	input_layer_1[0][0]
dense_2 (Dense)	(None, 16)	272	lambda[0][0]
softmax (Softmax)	(None, 16)	0	dense_2[0][0]
multiply (Multiply)	(None, 16)	0	lambda[0][0], softmax[0][0]
lstm_1 (LSTM)	(None, 64)	16,896	input_layer_1[0][0]
dense_3 (Dense)	(None, 64)	1,088	multiply[0][0]
concatenate (Concatenate)	(None, 128)	0	lstm_1[0][0], dense_3[0][0]
dense_4 (Dense)	(None, 64)	8,256	concatenate[0][0]
dense_5 (Dense)	(None, 1)	65	dense_4[0][0]

3.1.2 Fourier Transformed LSTM without Softmax Attention

The second model removes the softmax attention layer, opting to allow all frequency components to contribute with their original value to the prediction. This model tests whether the softmax attention, which emphasizes certain frequencies, is essential or if simpler attention mechanisms can still yield strong performance. This could be particularly useful for datasets where all frequency components are equally important or when the attention mechanism itself introduces noise.

Model Structure

- **LSTM Layer**: same as the previous model, capturing temporal dependencies.
- **Fourier Transform Layer**: same as before, used to capture frequency-domain information.
- **No Attention Mechanism**: the model processes all frequencies without emphasizing any specific ones.

- **Dense Layers**: the combined information from the time and frequency domains is passed through dense layers for the final prediction.

```
# Fourier Transformed LSTM without Softmax Attention
input_sales = Input(shape=(sequence_length, 1))

# LSTM Layer for time-domain analysis
x_time = LSTM(64, return_sequences=False)(input_sales)

# Fourier Transform Layer
x_freq = Lambda(fourier_transform_layer)(input_sales)

# Dense Layer for combined analysis
x_combined = Concatenate()([x_time, x_freq])
x_output = Dense(64, activation='relu')(x_combined)
x_output = Dense(1)(x_output) # Future sales prediction

model = Model(inputs=input_sales, outputs=x_output)
```

The result of this code is a model with the following structure:

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 30, 1)	0	-
lambda_1 (Lambda)	(None, 16)	0	input_layer_2[0][0]
dense_6 (Dense)	(None, 16)	272	lambda_1[0][0]
multiply_1 (Multiply)	(None, 16)	0	lambda_1[0][0], dense_6[0][0]
lstm_2 (LSTM)	(None, 64)	16,896	input_layer_2[0][0]
dense_7 (Dense)	(None, 64)	1,088	multiply_1[0][0]
concatenate_1 (Concatenate)	(None, 128)	0	lstm_2[0][0], dense_7[0][0]
dense_8 (Dense)	(None, 64)	8,256	concatenate_1[0][0]
dense_9 (Dense)	(None, 1)	65	dense_8[0][0]

3.1.3 Leaky ReLU Activation for Attention Scores

The third model explores the use of a Leaky ReLU activation function for the attention mechanism instead of the typical softmax or tanh functions. The Leaky ReLU function helps to account for negative values in attention scores, ensuring that negative attention scores can still have an influence. This model could be advantageous when certain negative frequencies or time periods play a crucial role in future predictions.

Model Structure

- **LSTM Layer**: similar to the previous models.
- **Fourier Transform Layer**: same Fourier transformation to extract frequency features.
- **Leaky ReLU Attention**: Leaky ReLU activation allows attention scores to handle negative values, which can reveal patterns that are lost with purely positive attention mechanisms.
- **Dense Layers**: the combined outputs from the LSTM and the Leaky ReLU attention layer are passed to dense layers for prediction.

```
# Fourier Transformed LSTM with Leaky ReLU Attention
input_sales = Input(shape=(sequence_length, 1))

# LSTM Layer for time-domain analysis
x_time = LSTM(64, return_sequences=False)(input_sales)

# Fourier Transform Layer
x_freq = Lambda(fourier_transform_layer)(input_sales)
```

```
# Leaky ReLU Attention
attention_scores = Dense(fft_length,
    activation=LeakyReLU(negative_slope=0.03))(x_freq)
x_freq_attended = Multiply()([x_freq, attention_scores])

# Dense Layer for combined analysis
x_combined = Concatenate()([x_time, x_freq_attended])
x_output = Dense(64, activation='relu')(x_combined)
x_output = Dense(1)(x_output) # Future sales prediction

model = Model(inputs=input_sales, outputs=x_output)
```

The result of this code is a model with the following structure:

Layer (type)	Output Shape	Param #	Connected to
input_layer_22 (InputLayer)	(None, 30, 1)	0	-
lambda_21 (Lambda)	(None, 16)	0	input_layer_22[0][0]
dense_86 (Dense)	(None, 16)	272	lambda_21[0][0]
softmax_9 (Softmax)	(None, 16)	0	dense_86[0][0]
multiply_21 (Multiply)	(None, 16)	0	lambda_21[0][0], softmax_9[0][0]
lstm_22 (LSTM)	(None, 64)	16,896	input_layer_22[0][0]
dense_87 (Dense)	(None, 64)	1,088	multiply_21[0][0]
concatenate_21 (Concatenate)	(None, 128)	0	lstm_22[0][0], dense_87[0][0]
dense_88 (Dense)	(None, 64)	8,256	concatenate_21[0][0]
dense_89 (Dense)	(None, 1)	65	dense_88[0][0]

3.1.4 Comparison

In this section, I will visually demonstrate the performance of each model, followed by a table presenting the MSE for each one.

It's important to note that the baseline and all four proposed models were trained using the same training, validation, and test sets.

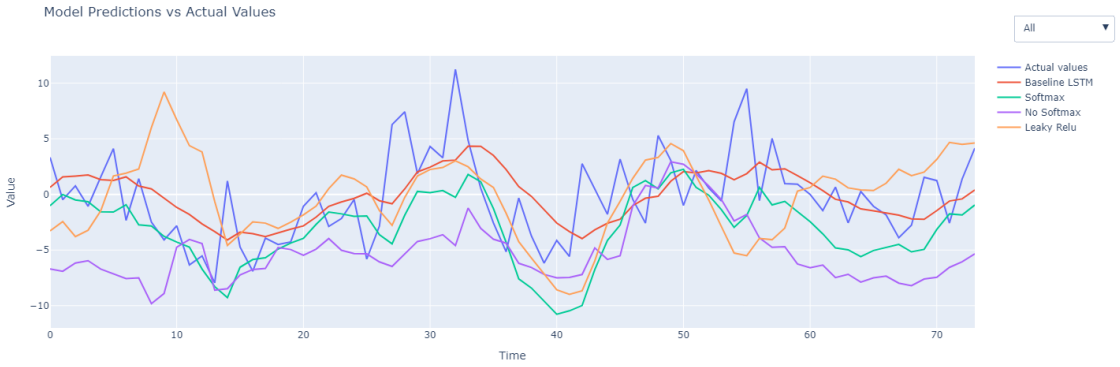


Figure 3.1: Comparison between v0 models and baseline

The obtained MSE table is:

Model	**MSE**
Baseline LSTM	11.458
Softmax	21.027
No Softmax	40.049
Leaky ReLU	27.358

Table 3.1: MSE values for different models, Version 0

The MSE table reveals that the Baseline LSTM model provides the best performance, with the lowest error (MSE = 11.458). Introducing the Softmax function leads to a noticeable increase in the error (MSE = 21.027). Removing the Softmax function further degrades performance, resulting in an even higher MSE of 40.049. The Leaky ReLU model shows a moderate improvement over the No Softmax variant, with an MSE of 27.358, but it still performs worse than the Baseline LSTM.

These results suggest that the Baseline LSTM remains the most effective model among those tested, and modifications such as Softmax and Leaky ReLU have not yielded better performance in terms of reducing error. Future work will focus on

further refining the models to achieve a lower MSE and potentially outperform the Baseline LSTM.

3.2 Version 1

Key Differences between Model 0 and Model 1

- **Data scaling:**
  - **Model 0:** No scaling transformation is applied to the data.
  - **Model 1:** Uses a *MinMaxScaler* to normalize the input data between 0 and 1. This is a significant improvement, as LSTMs are sensitive to data scale, and normalization can enhance model performance and convergence speed.
- **LSTM layer size:**
  - **Model 0:** Uses an LSTM layer with 64 units for temporal analysis.
  - **Model 1:** Increases the number of LSTM units to **128** and changes the activation function to **"relu"**. Increasing the units allows the model to capture more complexity in temporal dependencies, while the "relu" activation replaces the default "tanh", improving gradient handling by avoiding saturation.
- **Frequency network architecture:**
  - **Model 0:** The FFT output passes through a dense layer with a "tanh" activation.
  - **Model 1:** Changes the activation of the dense layer for attention to **"relu"**, which can improve performance, especially with normalized data.
- **Optimizer:**
  - **Model 0:** Uses the *Adam* optimizer.
  - **Model 1:** Changes the optimizer to *RMSprop*, which is often preferred for recurrent models like LSTMs, as it handles gradient oscillations better.

Expected Improvements

- The data normalization and increased LSTM capacity in **Model 1** should improve performance, especially when dealing with more complex variations in the data.
- Switching the optimizer from Adam to **RMSprop** should make gradient descent more stable in recurrent contexts, further enhancing the training process.

The modifications for each "subversion" are identical to those explained in the following chapters:

- Softmax: (3.1.1).
- No Softmax: (3.1.2).
- Leaky ReLU: (3.1.3).

Therefore, I will not reiterate the details of these modifications, as they are the same as in Version 0 [See chapter: (3)]. However, each modification now starts from the "subversion" Softmax of each Version (until Version 4, where Softmax will be replaced with Sigmoid).

3.2.1 Comparison

Here, we will compare our new models with the baseline to assess whether the modifications have improved the predictions.

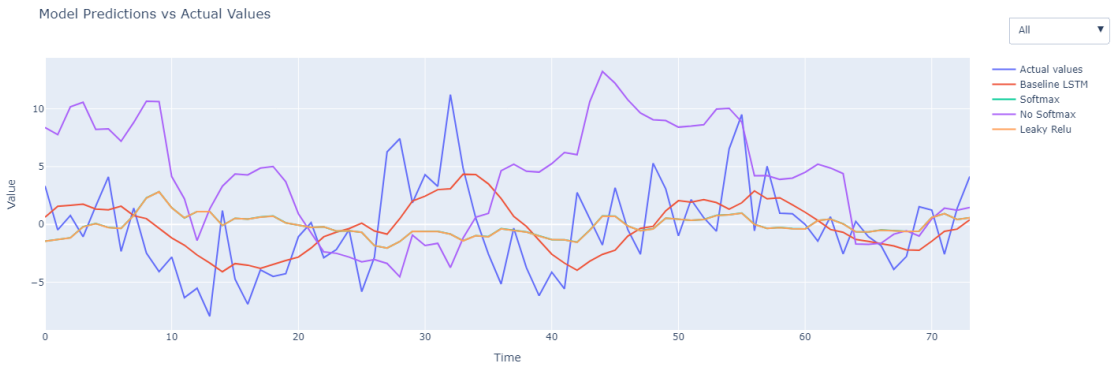


Figure 3.2: Comparison between v1 models and baseline

The obtained MSE table is:

Model	**MSE**
Baseline LSTM	11.458
Softmax	16.937
No Softmax	54.045
Leaky ReLU	16.941

Table 3.2: MSE values for different models, Version 1

The obtained MSE table demonstrates that the Baseline LSTM model continues to deliver the best performance, achieving the lowest error with an MSE of 11.458. This result confirms its superior accuracy compared to the other configurations. The model with the Softmax function shows an MSE of 16.937, which, although better than the previous result for this configuration, still lags behind the Baseline LSTM.

On the other hand, removing the Softmax function leads to a significant increase in error, with the MSE rising to 54.045. This highlights the important role the Softmax function plays in improving the model's predictions, as its absence results in much poorer performance.

The Leaky ReLU configuration performs almost identically to the Softmax version, with an MSE of 16.941, but it still fails to outperform the Baseline LSTM. These findings suggest that while Leaky ReLU can slightly improve performance compared to some configurations, it does not offer a significant advantage over the baseline.

Overall, the Baseline LSTM remains the best-performing model, and future work will focus on exploring further modifications to reduce the MSE and potentially surpass this baseline performance.

3.3 Version 2

Key Differences between Model 1 and Model 2

- **Data scaling:**
  - **Model 1:** Only normalizes the input using *MinMaxScaler* between 0 and 1, but does not apply any scaling operation on the targets (y).
  - **Model 2:** Normalizes both the input data (X) and the targets (y) using *MinMaxScaler*. This more comprehensive approach ensures that both input and target scaling are aligned, facilitating better optimization and model performance.
- **Data reshaping before scaling:**
  - **Model 1:** No explicit reshaping of the data is used before scaling.
  - **Model 2:** Data is reshaped into 1D vectors ("`reshape(-1, 1)`") before normalization, and then reverted to its original shape after scaling. This ensures the scaler correctly applies the transformation to the data.
- **Inverse scaling of predictions:**
  - **Model 1:** There is no inversion of the scaling applied to the predicted data, so predictions remain in the normalized scale.
  - **Model 2:** After prediction, the model reverses the scaling of the output data using "`scaler_y.inverse_transform`". This step is crucial for returning the data to its original scale and for accurate comparison between predictions and real values.
- **LSTM capacity and network structure:**
  - The network architecture of **Model 2** remains similar to **Model 1**, retaining 128 LSTM units, FFT-based frequency analysis, and an attention mechanism. The main difference is the handling of scaling, making the model more robust in practical contexts where data may vary greatly in scale.

- **Comparison in attention weights usage:**

  - **Model 1:** Uses *"relu"* for calculating attention scores on the frequency components, followed by a softmax to normalize the weights.
  - **Model 2:** Maintains the same attention configuration, using *"relu"* for the scores and *"softmax"* to normalize the weights. No significant differences in this aspect.
- **Prediction process:**
  - **Model 1:** The predictions are directly plotted after training without reversing the scaling.
  - **Model 2:** The predictions are reverted to the original scale using the scaler, providing a more interpretable and accurate comparison with the real data.

Similarities between Model 1 and Model 2

- **Model architecture:** Both models use an LSTM-based architecture for temporal analysis, FFT for frequency analysis, and an attention mechanism to enhance the relevance of frequency components.
- **Optimizer:** Both use *RMSprop* as the optimizer and apply *MSE* as the loss function.
- **Early stopping mechanism:** Present in both models to prevent overfitting.
- **Evaluation and visualization:** Both models predict and evaluate performance using MSE and plot both the loss values and predictions against real data.

Expected Improvements

- **Model 2** is expected to improve overall accuracy due to the normalization of both input and target data, and the subsequent inversion of scaling, providing predictions that are more accurate and comparable in the original scale.

3.3.1 Comparison

As before, in this section I'll show the differences between baseline and the other "subversion" of **"Version 2"**.

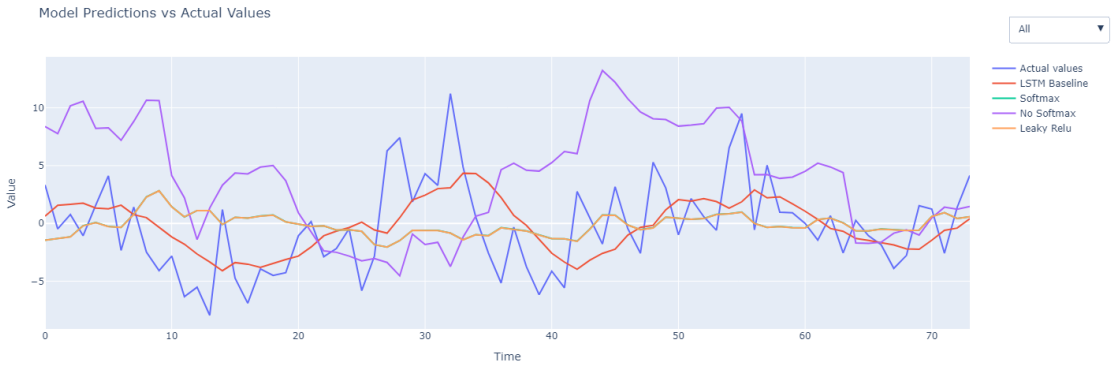


Figure 3.3: Comparison between v2 models and baseline

The obtained MSE table is:

Model	MSE
Baseline LSTM	11.458
Softmax	16.937
No Softmax	54.045
Leaky ReLU	16.941

Table 3.3: MSE values for different models, Version 2

The MSE table for Version 2 shows that the Baseline LSTM continues to provide the lowest error (MSE = 11.458), maintaining its status as the most effective model. The Softmax model, with an MSE of 16.937, shows some degradation in performance compared to the Baseline, but still offers better results than the other variants. On the other hand, the removal of the Softmax function results in a significant increase in error (MSE = 54.045), suggesting that the absence of this function negatively impacts the model's accuracy. Similarly, the Leaky ReLU model shows poor performance with an MSE of 16.941, similar to the Softmax.

3.4 Version 3

Key Differences between Model 2 and Model 3

- **Trainable Weights in FFT Attention:**
  - **Model 2:** Applies attention on frequency components using a `Dense(fft_length, activation=‘‘relu")` followed by a *Softmax* to normalize weights.
  - **Model 3:** Optimizes the attention mechanism by using a `Dense(1, activation=‘‘relu")` for calculating attention scores, followed by `Flatten()` and *Softmax* over the entire frequency dimension. This change allows a finer control over the frequency components, enhancing the model's focus on relevant frequencies.
- **Flattening of Frequency Features:**
  - **Model 2:** Does not flatten the frequency features after applying attention weights.
  - **Model 3:** After applying attention weights on frequency components, the resulting features are flattened using `Flatten()`. This step facilitates a more seamless integration with temporal features from the LSTM, improving the model's capability to process combined data.
- **Dense Layer Structure:**
  - **Model 2:** Uses a dense layer with 64 units after the attention mechanism and combines the temporal and frequency results.
  - **Model 3:** Retains the same architecture but adds a flattening layer after attention. This modification might enhance the model's ability to process frequency data more effectively.
- **Activation in Attention Mechanism:**
  - **Model 2:** Uses *relu* to calculate attention scores.
  - **Model 3:** Keeps *relu* for calculating attention scores, but uses a more compact architecture with a single dense node, which streamlines attention computation.

- **Reduced Attention Complexity:**
  - **Model 3:** Reduces the complexity of attention weights by using a dense layer with a single node, focusing attention on relevant frequency components with greater precision.

Similarities between Model 2 and Model 3

- **General Architecture:** Both models follow a similar architecture combining an LSTM branch for temporal analysis and an FFT-based branch for frequency analysis.
- **Data Scaling:** Both normalize input and target data using *MinMaxScaler* before training.
- **Optimizer and Loss Function:** Both use *RMSprop* as the optimizer and *MSE* as the loss function.
- **Early Stopping Mechanism:** Both models apply early stopping with a patience of 10 epochs to avoid overfitting.

Expected Improvements

- **Improved Attention Mechanism:** The refined attention in **Model 3** is expected to provide more precise selection of relevant frequency components, improving prediction accuracy.
- **Reduced Complexity:** The simpler attention structure using a `Dense(1)` for attention scores reduces computational complexity and potentially mitigates the risk of overfitting.

3.4.1 Comparison

As before, this section will present the differences between baseline and the different "subversions" of **Version 3**.

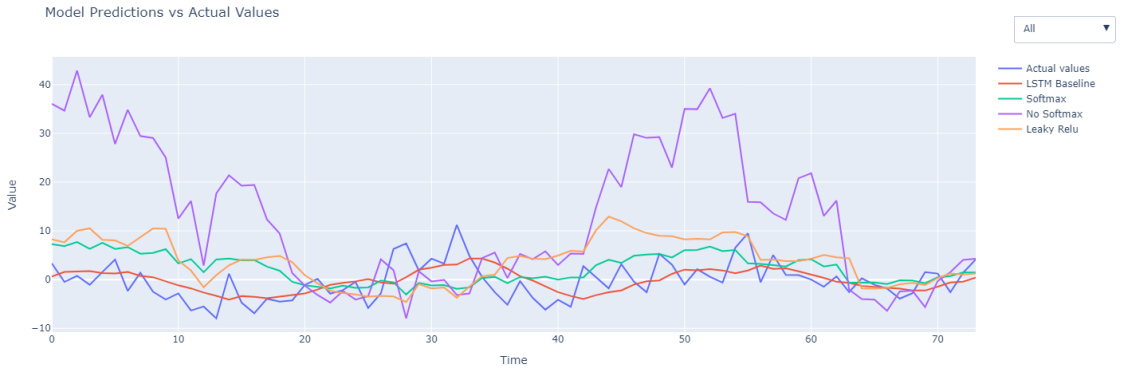


Figure 3.4: Comparison between Version 3 models and baseline

The obtained MSE table is:

Model	**MSE**
Baseline LSTM	11.458
Softmax	29.334
No Softmax	373.295
Leaky ReLU	51.933

Table 3.4: MSE values for different models, Version 3

The MSE table for Version 3 shows a notable decline in performance across all models compared to previous versions. The Baseline LSTM remains the most effective model with an MSE of 11.458, consistent with prior results. However, the Softmax model exhibits a significant increase in error, reaching an MSE of 29.334, which is considerably worse than in Version 2. The removal of the Softmax function leads to an even larger degradation in performance, with the MSE skyrocketing to 373.295. Similarly, the Leaky ReLU model shows poor performance, although it is better than the No Softmax version, with an MSE of 51.933.

3.5 Version 4

Key Differences between Model 3 and Model 4

- **Frequency Filtering in FFT Layer:**
  - **Model 3:** Uses the Fourier Transform without explicit selection of frequency components.
  - **Model 4:** Introduces a range of periods (minimum 10, maximum 100), filtering the frequency components corresponding to this period range. This allows the model to focus on more relevant frequencies for long-term forecasting, ignoring high-frequency noise or insignificant variations.
- **Sigmoid Activation in Attention Mechanism:**
  - **Model 3:** Uses *Softmax* to normalize attention weights, ensuring the sum of weights equals 1.
  - **Model 4:** Replaces *Softmax* with ***Sigmoid*** for attention weights over frequency components. *Sigmoid* allows independent importance weighting for each frequency, without imposing a unit sum constraint. This is especially useful in conjunction with the frequency filtering mechanism.
- **Exclusion of Zero Frequency Component:**
  - **Model 4:** Excludes the zero frequency (DC component) from the FFT magnitude spectrum to prevent division by zero errors and avoid non-oscillatory component influence, ensuring the attention mechanism focuses on meaningful oscillatory frequencies.
- **Architecture Adjustments:**
  - **Model 3:** Combines time and frequency components without period filtering.
  - **Model 4:** Adds a reshape step after applying attention weights to match the expected frequency shapes, ensuring correct application of the attention mechanism.

Similarities between Model 3 and Model 4

- **Temporal Branch with LSTM:** Both models retain the same architecture for temporal analysis, using an LSTM with 128 units and *relu* activation.
- **Data Scaling:** Both models normalize input and target data using *MinMaxScaler* before training, with predicted results inverse scaled to the original range.
- **Overall Architecture:** Both models combine temporal and frequency features, and use dense layers for the final output.

Expected Improvements

- **Frequency Filtering:** By focusing on a specific range of relevant frequencies, the model reduces complexity and ignores noise or variations not useful for long-term forecasting.
- **Sigmoid Attention Weights:** Replacing *Softmax* with *Sigmoid* allows the model to assign independent importance to each frequency without constraints, potentially improving performance when some frequencies are not correlated.

3.5.1 Comparison

As before, this section presents the differences between the baseline and the different "subversions" of **Version 4**.

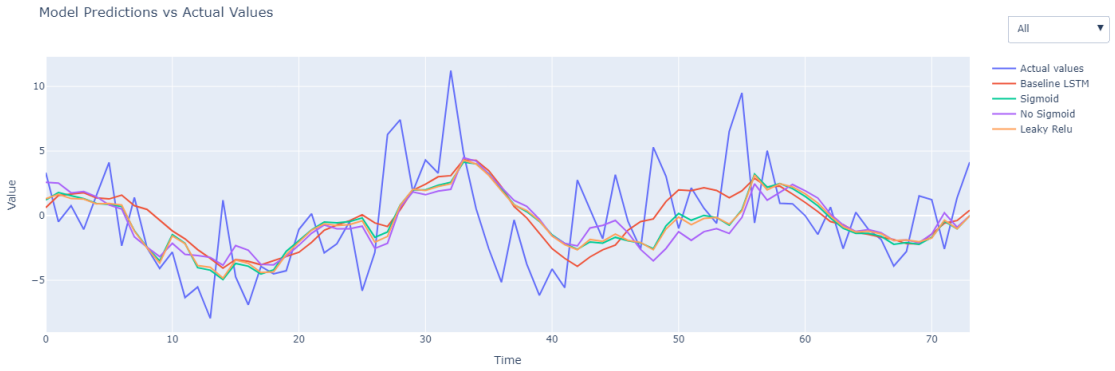


Figure 3.5: Comparison between Version 4 models and baseline

The obtained MSE table is:

Model	**MSE**
Baseline LSTM	11.458
Sigmoid	12.085
No Sigmoid	13.336
Leaky ReLU	12.183

Table 3.5: MSE values for different models, Version 4

The MSE table for Version 4 shows that the Baseline LSTM continues to perform the best, with an MSE of 11.458. The Sigmoid model, with an MSE of 12.085, exhibits a slight increase in error compared to the baseline, while the No Sigmoid version has a higher MSE of 13.336. The Leaky ReLU model performs in between these values, with an MSE of 12.183.

These results indicate that the attention mechanism changes and the introduction of frequency filtering in Version 4 are effective in enhancing the selection of relevant frequency components. While some models show a slight increase in error, the improvements suggest that the model is better equipped to focus on meaningful oscillatory patterns, filtering out noise and irrelevant high-frequency variations, which helps with long-term prediction accuracy.

3.6 Version 5

Here is the analysis of **Model 5** compared to **Model 4**, with a particular focus on the newly introduced changes.

Key Differences between Model 4 and Model 5

- **Introduction of Dropout:**
  - **Model 4:** Does not use dropout, which can expose the model to the risk of overfitting, especially with smaller datasets.
  - **Model 5:** Introduces `Dropout(0.3)` at several points in the model, including the temporal and frequency branches, as well as in the final dense networks. Dropout is a regularization technique that randomly deactivates a percentage of the nodes during training, forcing the model to generalize better and reducing overfitting.
- **Dropout in Temporal and Frequency Branches:**
  - **Temporal Branch:** After the LSTM output, a `Dropout(0.3)` layer is added to improve the regularization of the temporal output.
  - **Frequency Branch:** After applying attention weights and flattening the frequency components, `Dropout(0.3)` is applied to reduce overfitting to specific frequency characteristics.
  - **Final Dense Layer:** After the final dense layer, which combines the temporal and frequency features, another `Dropout(0.3)` is applied.
- **Purpose of Dropout:**
  - The **dropout** in this model serves to reduce overfitting, which may have been present in previous versions without regularization. The model trains in a more robust manner, learning to generalize better rather than overly adapting to the training data.
- **Expected Performance:**
  - The addition of dropout is expected to help prevent a performance drop when the model is tested on new data. Overfitting is a common issue

in complex models like those with LSTM and frequency branches, and dropout is one of the main techniques to counteract it.

Similarities between Model 4 and Model 5

- **Model architecture:** Both models use the same basic structure with temporal (LSTM) and frequency (FFT) branches, combined through an attention mechanism on filtered frequency components.
- **Attention mechanism:** Both continue to use **Sigmoid activation** for the attention weights on the frequency components, which allows assigning importance to specific relevant frequencies.
- **Loss function and optimization:** Both use **RMSprop** as the optimizer and **MSE** as the loss function.
- **Early stopping:** Both models maintain the **early stopping** mechanism to prevent overfitting during training, stopping the process if the validation improvement halts.

Expected Improvements

- **Better Generalization: Model 5**, with the introduction of dropout, is expected to show greater robustness when applied to new data. While previous models may have achieved good performance on training and validation data, they may have suffered from overfitting. Dropout reduces this tendency, allowing the model to perform better on test data.
- **Reduced Risk of Overfitting:** Thanks to the regularization introduced by dropout, the model should be less inclined to "memorize" the training data and more likely to generalize well on test or future data.

3.6.1 Comparison

As before, this section presents the differences between the baseline and the different "subversions" of **Version 5**.

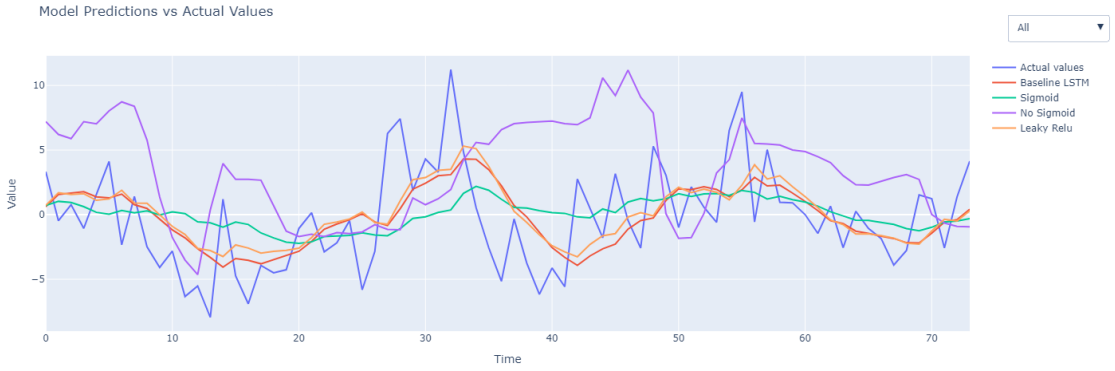


Figure 3.6: Comparison between Version 5 models and baseline

The obtained MSE table is:

Model	**MSE**
Baseline LSTM	11.458
Sigmoid	13.747
No Sigmoid	38.183
Leaky ReLU	11.376

Table 3.6: MSE values for different models, Version 5

The MSE table for Version 5 illustrates the performance of various models, with the Baseline LSTM once again achieving the lowest error, recording an MSE of 11.458. This reinforces its reliability and effectiveness in comparison to the modified models in this version.

The Sigmoid model, which introduced changes to the attention mechanism, shows an increased error with an MSE of 13.747. This suggests that while the independent weighting of frequency components using Sigmoid can be beneficial in theory, it may

require further adjustments to compete with the baseline performance effectively.

The No Sigmoid version exhibits a significantly higher error, with an MSE of 38.183. This sharp increase implies that removing the Sigmoid activation for attention weights leads to a substantial degradation in model performance, likely due to the lack of proper normalization and weighting of the frequency components.

Lastly, the Leaky ReLU model performs the best among the modified models, with an MSE of 11.376, coming very close to the Baseline LSTM. This indicates that the use of Leaky ReLU activation improves performance over both the Sigmoid and No Sigmoid models and nearly matches the effectiveness of the Baseline LSTM.

	Mean Squared Error				
	Softmax	***No Softmax***	***LeakyReLu***	***Sigmoid***	***No Sigmoid***
Version 0	21.027	40.049	27.358	/	/
Version 1	16.937	54.045	16.941	/	/
Version 2	16.937	54.045	16.941	/	/
Version 3	29.334	373.295	51.993	/	/
Version 4	/	/	12.183	12.085	13.336
Version 5	/	/	11.376	13.747	38.183

3.6.2 Analysis and Observations

- **Performance of the Baseline LSTM:** The baseline LSTM model maintains a strong performance with an MSE of 11.458, slightly outperforming the Leaky ReLU model (MSE 11.376). This reinforces the robustness of the baseline architecture, showing that while modifications can yield comparable results, they do not always guarantee significant improvements.
- **Leaky ReLU vs. Sigmoid:** The Leaky ReLU model achieved better performance than the Sigmoid model, with MSE values of 11.376 and 13.747, respectively. This suggests that Leaky ReLU is better suited for this model, offering smoother gradient propagation and helping prevent issues like vanish-

ing gradients that could occur with Sigmoid activations.

- **No Sigmoid Variant Underperformance:** The No Sigmoid version demonstrated a significant performance drop with an MSE of 38.183, indicating that removing the activation function from the attention mechanism leads to poor weighting of the frequency components and ineffective model generalization.
- **Baseline Effectiveness and Modifications:** The baseline LSTM continues to perform very well despite various model modifications. This suggests that the simpler baseline model has inherent strengths that are difficult to surpass without more advanced techniques or fine-tuning.

4. Improvement from the Best

In the previous chapters, we explored various model modifications and their respective performances, identifying the Leaky ReLU version as the best-performing model with an MSE of 11.376, closely matching the baseline LSTM (MSE 11.458). While the Leaky ReLU model shows promise, the objective of this chapter is to further refine this architecture, aiming to outperform the baseline model and achieve improved generalization on unseen data.

To accomplish this, we will build upon the current best combination of parameters and techniques, specifically by:

- Modifying the temporal branch of the model to be bidirectional, allowing for better exploitation of both past and future information in the sequence.
- Increasing the variable `sequence_length`
- Increasing the number of input signals with common features to enhance the model's capacity to capture underlying patterns.

4.1 What is a Bi-LSTM and Why Use It

A Bidirectional Long Short-Term Memory (BiLSTM) network is a variant of the standard LSTM that processes data in both forward and backward directions. While traditional LSTMs capture dependencies in a unidirectional manner, processing input only from past to future, a BiLSTM incorporates two separate LSTM layers: one processes the sequence from the past to the future (forward direction), and the other processes it in reverse, from the future to the past (backward direction).

This bidirectional architecture allows the network to learn not only from preceding data points but also from future data points in the sequence. In tasks where the context surrounding a particular time step is crucial, such as time series prediction, this dual perspective can enhance the model's understanding of temporal dependencies.

The decision to implement a bidirectional LSTM in the temporal branch of the model

stems from the expectation that it will improve the model's ability to learn from the entire input sequence, leading to better generalization and performance. Specifically, in our task, where signals are influenced by both past and future dynamics, this approach could provide significant benefits.

4.1.1 Starting with a BiLSTM

The first enhancement I aim to introduce is replacing the LSTM with a BiLSTM in the time domain branch. This modification only requires a simple change to the code, specifically by updating the line defining the time branch as follows:

```
# Time domain branch using Bidirectional LSTM
x_time = Bidirectional(LSTM(128, activation='relu',
    return_sequences=False))(input_sales)
```

Results

The model using a BiLSTM produced the following predictions:

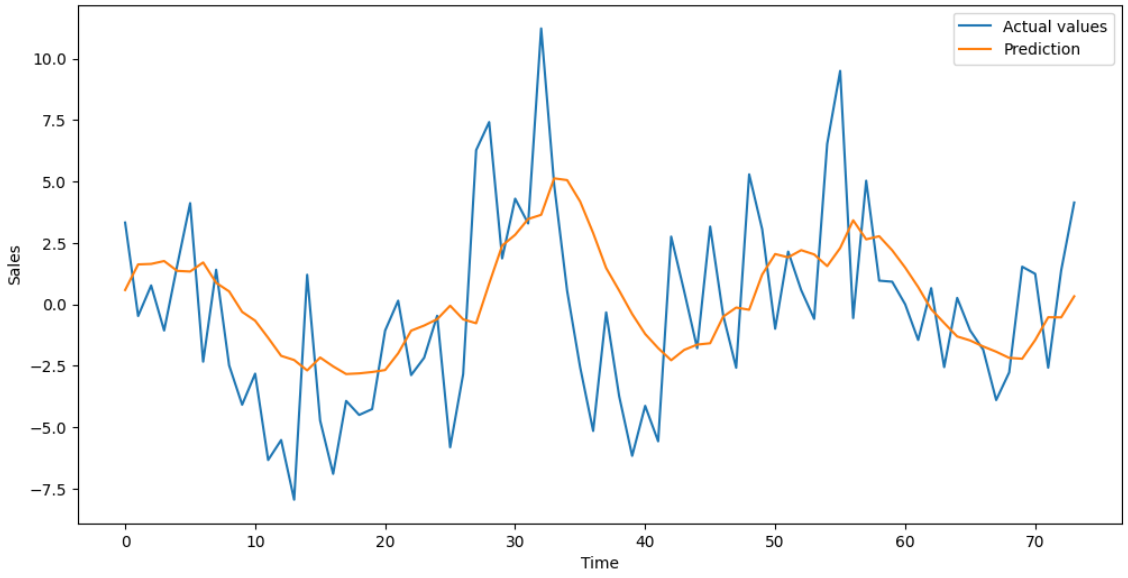


Figure 4.1: BiLSTM Model Predictions

The predictions look promising, but has there been an improvement?

Model	MSE
Baseline LSTM	11.458
LeakyReLU v5	11.376
LeakyReLU BiLSTM	11.867

Table 4.1: Comparison of Models Mean Squared Error (MSE)

What we were able to see is that by simply modifying the temporal branch no improvement were introduced in the time series predictions. In the next section I'll try to increase the sequence length when generating the input.

4.2 Why Increase the Sequence Length

Increasing the `sequence_length` refers to extending the number of time steps that the model considers as input. This can be particularly beneficial when dealing with temporal data, as it allows the model to capture longer-term dependencies and patterns within the data.

By increasing the sequence length, the model has access to a broader context, which is crucial for accurately modeling complex time series where past observations may influence the future over extended periods. This can improve the model's ability to understand and predict trends that span over a longer range of time steps. For instance, certain periodic behaviors or delayed effects in the data may only be captured when the model is able to consider a larger portion of the input sequence.

In our case, increasing the sequence length allows the model to have a more comprehensive view of the signal's behavior, which is particularly useful when signals exhibit long-term dependencies or trends. It helps the model to better identify recurring patterns and relationships between past and future data points, leading to more accurate and robust predictions. This adjustment aims to improve both

short-term and long-term forecasting capabilities, reducing the likelihood of missing important temporal dynamics.

To assess whether increasing the sequence length enhances the model's performance, we began with an initial sequence length of 30. We then doubled it, and if no notable improvements are observed, we will test by tripling the length.

Model	**MSE**
Baseline LSTM	11.458
LeakyReLU BiLSTM (sequence = 60)	10.096
LeakyReLU BiLSTM (sequence = 90)	19.136

Table 4.2: Comparison of Models' Mean Squared Error (MSE)

From the results, we observe that increasing the sequence length from 30 to 60 yields a reduction in MSE, indicating improved model performance. However, further increasing the sequence length to 90 results in a higher MSE, suggesting that longer sequences may introduce unnecessary complexity or noise, which could hinder model performance. These findings suggest that an optimal sequence length exists for this dataset, and increasing it beyond this point may not be beneficial.

4.3 Why Use Multiple Signals to Train the Model

Incorporating multiple signals with shared characteristics, such as common noise and trends, can significantly enhance the model's ability to generalize and capture underlying patterns. When signals share a common noise component, the model has an opportunity to learn how to distinguish between noise and meaningful signal variations. This leads to improved robustness in predictions, as the model becomes better equipped to handle noisy inputs during both training and testing.

Moreover, multiple signals often exhibit common trends or underlying patterns, and training the model on these signals allows it to capture these shared dynamics. By

leveraging the relationships between different signals, the model can learn more generalized representations of the data, which in turn helps in improving the predictive power across all signals.

In our case, using multiple signals will allow the model to exploit the inherent correlations between them, which can enhance its capacity to identify and model the true signal trends, reducing overfitting and increasing the model's performance on unseen data. This is particularly useful when signals are affected by similar environmental or system-wide factors, allowing the model to better isolate the trend from the noise.

4.4 Input Generation

As previously mentioned, one of the first improvements I aim to implement is introducing different input data to the model while continuing the training on the same underlying patterns. The goal is for both the Fourier transformation and the LSTM to identify common structures within each signal, leading to better overall predictions.

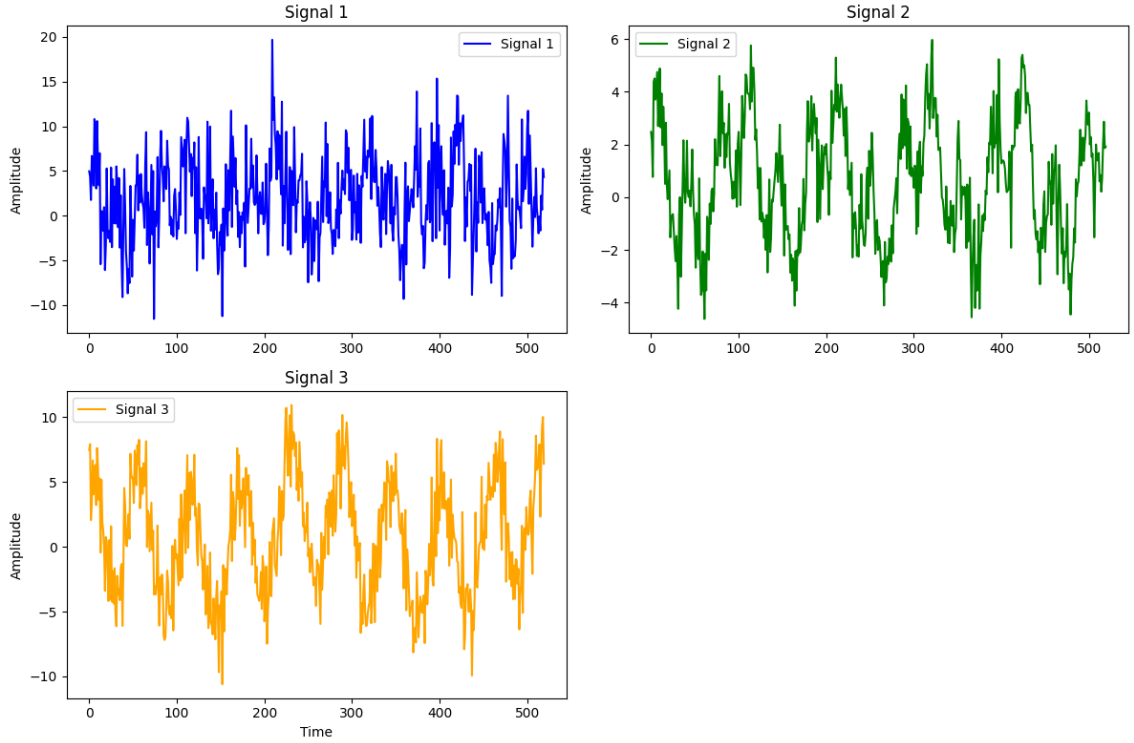
To achieve this, I created the following Python code:

```
res = 520
common_noise_1 = noise(res, 1)
common_noise_2 = noise(res, 2)
common_trend_1 = trend(res)
common_trend_2 = trend(res, c1 = 2)

signal_1 = sine_wave + common_trend_1 + common_noise_1 + common_trend_2 +
    common_noise_2
signal_2 = wave(15, res, 2) + wave(5, res, 2, np.cos) + common_trend_1 +
    common_noise_1
signal_3 = wave(9, res, 5, np.cos) + wave(2, res, 2, np.cos) +
    common_trend_1 + common_noise_1 + common_trend_2 + common_noise_2
```

As shown, there are some shared components across all the signals, such as the common trend and noise. This shared structure is intended to help the model recognize patterns and improve its predictive capabilities.

The resulting signals are:



The approach I followed to work with the LSTM baseline and my model is as follows:

1. Train on the first signal
2. Train on the second signal
3. Train on the third signal

After completing this training process, I proceeded with the prediction phase, aiming to observe whether the model can effectively capture shared frequencies.

Here are the results I obtained:

	Mean Squared Error		
	LSTM(seq=30)	***BiLSTM(seq=30)***	***BiLSTM(seq=60)***
Signal 1	23.666	16.619	20.079
Signal 2	2.835	2.009	1.458
Signal 3	6.468	7.333	9.630

Table 4.3: Mean Squared Error for Different Models and Sequences

It can be observed that the models incorporating frequency analysis improved the prediction of the signal in most cases. This suggests that our hypothesis of finding common patterns in the frequencies may be correct.

It could be useful to see if this suggestion is true by adding even more complex signals, each one sharing some common parts.

I create ten different signals in this way:

```
signal_1 = sine_wave + common_trend_1 + common_noise_1 + common_trend_2 +
    common_noise_2
signal_2 = wave(15, resolution, 2) + wave(5, resolution, 2, np.cos) +
    noise(resolution, 3) + trend(resolution) + common_trend_1 +
    common_noise_1
signal_3 = wave(9, resolution, 5, np.cos) + wave(2, resolution, 2,
    np.cos) + noise(resolution, 2) + trend(resolution) + common_noise_1 +
    common_trend_2 + common_noise_2
signal_4 = wave(10, resolution, 4) + wave(3, resolution, 6, np.sin) +
    wave(8, resolution, 2, np.cos) + noise(resolution, 1) +
    trend(resolution) + common_trend_1 + common_noise_1
signal_5 = wave(6, resolution, 3, np.cos) + wave(11, resolution, 2) +
    wave(7, resolution, 5) + noise(resolution, 2) + trend(resolution,
    c1=1.5) + common_noise_1 + common_trend_2
```

```
signal_6 = wave(12, resolution, 4) + wave(4, resolution, 7, np.cos) +
    wave(9, resolution, 3) + noise(resolution, 3) + trend(resolution,
    c1=1) + common_trend_1 + common_noise_2
signal_7 = wave(8, resolution, 2) + wave(5, resolution, 9, np.sin) +
    wave(3, resolution, 4, np.cos) + noise(resolution, 1) +
    trend(resolution) + common_trend_2 + common_noise_1
signal_8 = wave(10, resolution, 6) + wave(2, resolution, 3, np.sin) +
    wave(7, resolution, 5) + noise(resolution, 2) + trend(resolution,
    c1=1.8) + common_trend_1 + common_trend_2 + common_noise_1
signal_9 = wave(14, resolution, 1, np.cos) + wave(4, resolution, 10) +
    wave(6, resolution, 2) + noise(resolution, 3) + trend(resolution,
    c1=2.2) + common_noise_2 + common_trend_1
signal_10 = wave(5, resolution, 8) + wave(3, resolution, 4, np.sin) +
    wave(9, resolution, 6, np.cos) + noise(resolution, 2) +
    trend(resolution, c1=1.3) + common_trend_2 + common_noise_1 +
    common_noise_2

signals = [signal_1, signal_2, signal_3, signal_4, signal_5, signal_6,
    signal_7, signal_8, signal_9, signal_10]
```

Now we retry to train our models on all this signal and we watch if something better happen. Here are reported the obtained results:

	Mean Squared Error		
	LSTM(seq=30)	***BiLSTM(seq = 30)***	***BiLSTM(seq=60)***
Signal 1	26.539	21.468	23.371
Signal 2	3.920	3.327	4.054
Signal 3	6.566	7.837	5.780
Signal 4	3.018	3.194	3.815
Signal 5	6.571	7.172	6.268
Signal 6	15.061	15.592	14.869
Signal 7	2.854	3.754	3.505
Signal 8	7.553	7.900	7.445
Signal 9	15.131	15.199	16.190
Signal 10	12.124	13.580	12.251

The results indicate that the BiLSTM + Fourier models generally show some improvement over the LSTM baseline in several cases. However, the improvements are not consistent across all signals, with some cases (e.g., Signal 3, Signal 6, and Signal 9) showing mixed results or even higher errors for the BiLSTM + Fourier models compared to the LSTM.

The use of different sequence lengths (30 vs 60) produced variable outcomes: longer sequences sometimes led to better predictions (e.g., Signal 3) but occasionally increased the error (e.g., Signal 2 and Signal 9). This suggests that the choice of sequence length can significantly affect the model's ability to capture the patterns in the signals.

Overall, while the BiLSTM + Fourier approach does demonstrate potential benefits, the results indicate variability in performance across different signals, highlighting the need for further optimization or model adaptation based on the specific characteristics of each signal.

Here are reported some prediction representation:

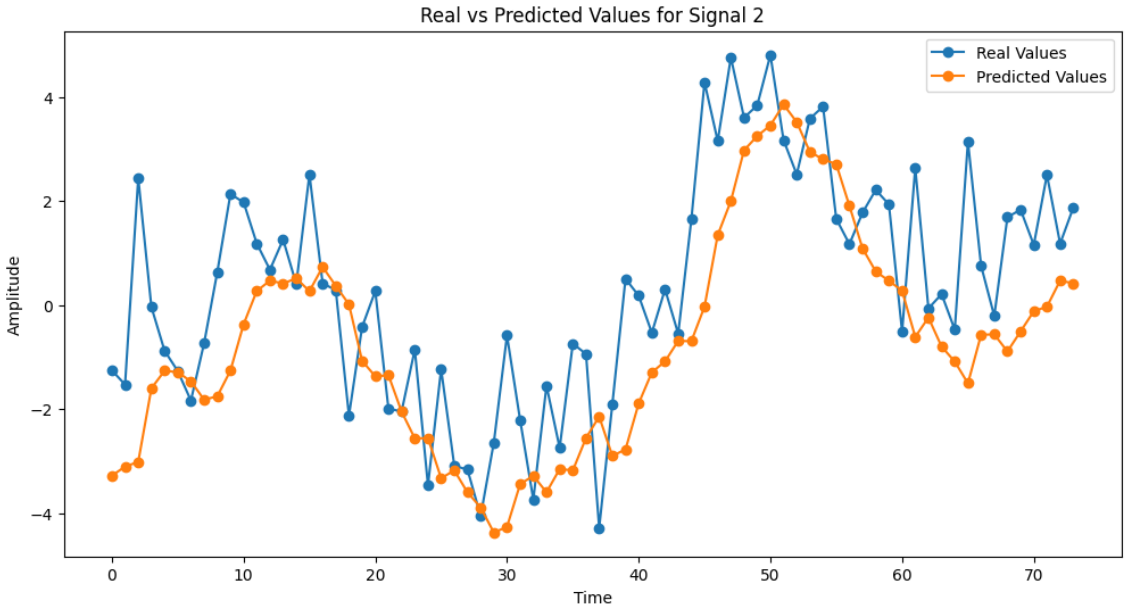


Figure 4.2: BiLSTM + Fourier prediction (sequence lenght = 60)

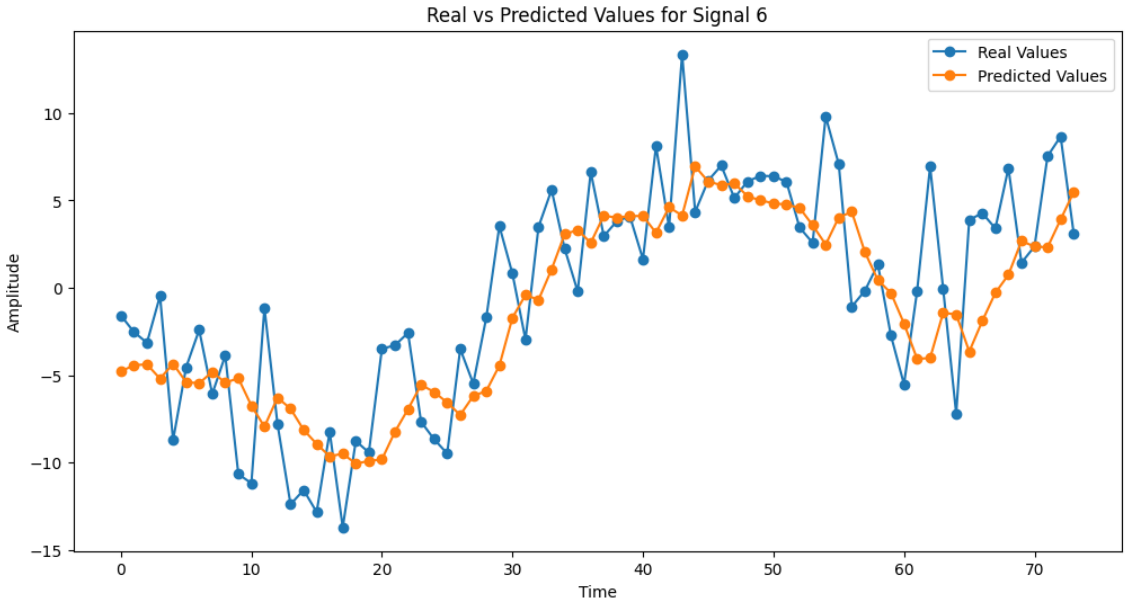


Figure 4.3: BiLSTM + Fourier prediction (sequence lenght = 60)

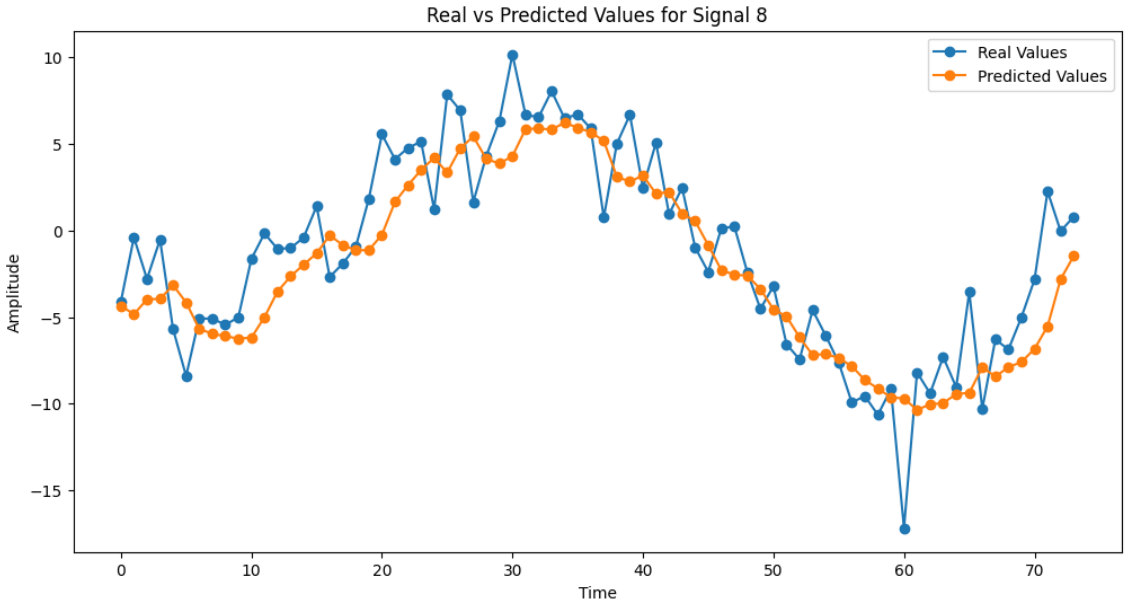


Figure 4.4: BiLSTM + Fourier prediction (sequence lenght = 60)

5. Adding More Information to the Signal

With the objective of improving the prediction, we could enhance the signal by adding additional information. The main idea is to create a new branch that incorporates the average values from the week, the past two weeks and the last month. This branch will process these averages through a simple dense layer, and its output will then be concatenated with the outputs from the temporal and frequency branches.

5.1 Data Preparation

The data preparation process involves generating features from the original signal to enhance the prediction accuracy. We generate two main types of features: lag features and rolling average features.

The following steps describe the data preparation process:

1. **Generating Lag Features**: Lag features are created by shifting the original time series data backward by a specified number of steps. Given a sequence length L, lag features $y_{-1}, y_{-2}, \ldots, y_{-L}$ are generated for each time step. This step allows the model to incorporate past values of the signal as input.
2. **Calculating Rolling Averages**: Rolling averages are calculated over different window sizes to capture trends over various periods. In this implementation, we compute the rolling averages over three different time spans:
   - **Weekly Average**: The average over the past 7 days.
   - **Biweekly Average**: The average over the past 14 days.
   - **Monthly Average**: The average over the past 30 days.

   These rolling averages help to smooth out short-term fluctuations and highlight longer-term trends in the data.

3. **Combining the Features**: The lag features and rolling averages are combined into separate data tables. The lag features are represented in a matrix where each column corresponds to a different lag, while the rolling averages form another matrix where each column corresponds to a different time window. Any rows containing `NaN` values due to shifting or rolling operations are removed to ensure clean input data.
4. **Preparing the Target Variable**: The target variable, which is the original time series shifted backward by the sequence length, is aligned with the generated features. This ensures that the input features are paired with the corresponding target values for supervised learning.

The final prepared dataset consists of two feature tables (lag features and rolling averages) and the target variable. These features are then fed into the three branches of the model, enabling the network to learn from both the original temporal data and the additional information provided by the moving averages.

5.2 Dense Layer

The model was initially designed with only two branches: a temporal branch and a frequency branch based on Fourier Transform. In this section, we extend the model by introducing a new branch for moving averages. The updated model structure includes three main branches:

1. **Temporal Branch**: This branch processes the original signal using a bidirectional LSTM.
2. **Frequency Branch**: This branch applies a Fourier Transform to the input signal, followed by an attention mechanism.
3. **Moving Averages Branch**: This newly added branch processes the moving averages from the past week and past two weeks.

The output from these branches is then concatenated and passed through a dense layer to make the final prediction.

As done in section (3.1) I decided to visualize the structure of the new model using the `summary` command. The result is:

Layer (type)	Output Shape	Param #	Connected to
input_layer_50 (InputLayer)	(None, 30, 1)	0	-
reshape_50 (Reshape)	(None, 30)	0	input_layer_50[0][0]
lambda_25 (Lambda)	(None, 16)	0	reshape_50[0][0]
dense_113 (Dense)	(None, 16)	272	lambda_25[0][0]
flatten_64 (Flatten)	(None, 16)	0	dense_113[0][0]
activation_25 (Activation)	(None, 16)	0	flatten_64[0][0]
reshape_51 (Reshape)	(None, 16)	0	activation_25[0][0]
multiply_25 (Multiply)	(None, 16)	0	lambda_25[0][0], reshape_51[0][0]
flatten_65 (Flatten)	(None, 16)	0	multiply_25[0][0]
input_layer_51 (InputLayer)	(None, 3, 1)	0	-
dropout_105 (Dropout)	(None, 16)	0	flatten_65[0][0]
dense_115 (Dense)	(None, 3, 64)	128	input_layer_51[0][0]
bidirectional_25 (Bidirectional)	(None, 256)	133,120	input_layer_50[0][0]
dense_114 (Dense)	(None, 64)	1,088	dropout_105[0][0]
flatten_66 (Flatten)	(None, 192)	0	dense_115[0][0]
dropout_104 (Dropout)	(None, 256)	0	bidirectional_25[0][0]
dropout_106 (Dropout)	(None, 64)	0	dense_114[0][0]
dropout_107 (Dropout)	(None, 192)	0	flatten_66[0][0]
concatenate_21 (Concatenate)	(None, 512)	0	dropout_104[0][0], dropout_106[0][0], dropout_107[0][0]
dense_116 (Dense)	(None, 64)	32,832	concatenate_21[0][0]
dropout_108 (Dropout)	(None, 64)	0	dense_116[0][0]
dense_117 (Dense)	(None, 1)	65	dropout_108[0][0]

Figure 5.1: Structure of the new model

5.3 Attention Mechanism

Just for trying, before showing the results, I decided to create another model with a different structure. In this new model I added another an attention mechanism for the moving averages. The input moving averages are first reshaped, and then attention scores are computed using a dense layer with a Leaky ReLU activation function. The attention scores are converted to attention weights via a sigmoid activation and are applied to the input moving averages before being processed by a dense layer.

5.4 Comparison of Results

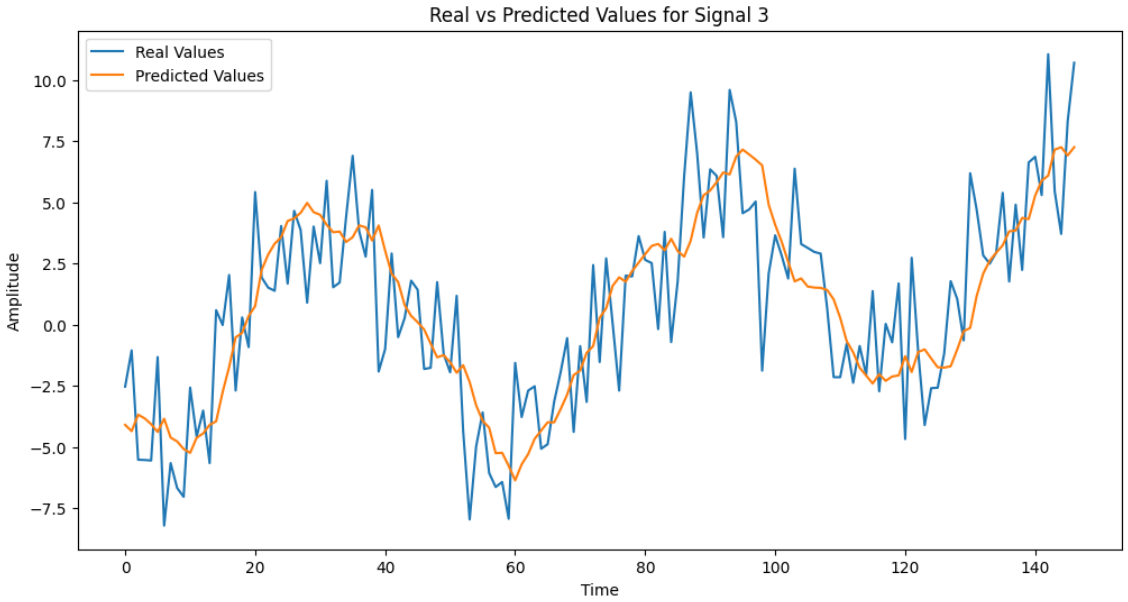
The table below presents a comparison of the Mean Squared Error (MSE) for different signals, with and without the attention mechanism applied in the moving averages branch. The results indicate that introducing the attention mechanism does not consistently improve the model's performance across all signals.

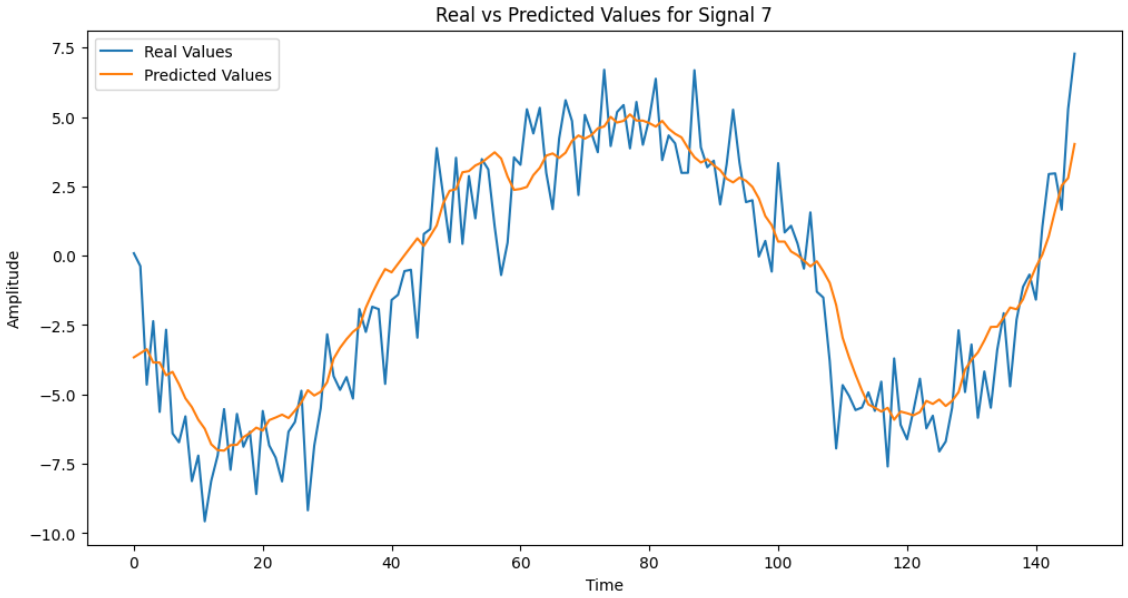
For some signals, such as Signal 3, Signal 5, and Signal 7, the attention mechanism leads to a lower MSE, suggesting that the model benefits from dynamically weighting the features in these cases. However, for other signals like Signal 1 and Signal 6, the attention mechanism results in a higher MSE, indicating that it may not always effectively enhance the predictive capability.

Overall, the mixed results highlight that while the attention mechanism can be beneficial for specific cases, it may introduce complexity without significant performance gains for others.

	Mean Squared Error	
	No attention mechanism	***Attention mechanism***
Signal 1	35.614	37.080
Signal 2	2.374	2.544
Signal 3	7.048	6.288
Signal 4	3.593	3.483
Signal 5	7.521	6.684
Signal 6	16.317	16.909
Signal 7	3.475	2.853
Signal 8	8.909	8.107
Signal 9	14.161	13.590
Signal 10	12.903	12.895

Let's see some plots, here are reported the prediction for **signal 3** and **signal 7**




Real vs Predicted Values for Signal 7
Real Values
Predicted Values
Amplitude
Time
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0
20
40
60
80
100
120
140

6. Everything Seems to Work Fine, but Is It Real?

So far, I have been working with complex signals, and our model appears to perform quite well on these. But are the results as reliable as they seem? To investigate this, let's simplify the signals and examine a few key considerations.

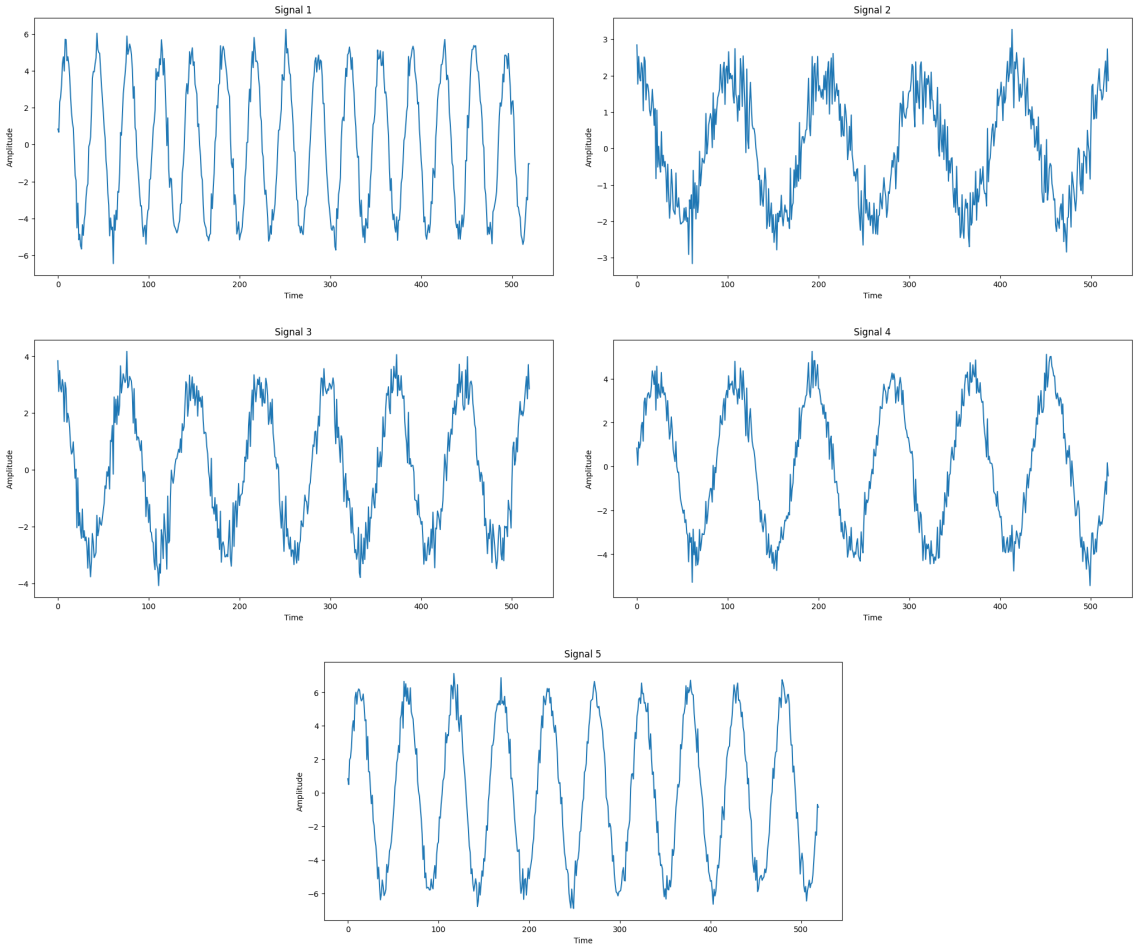
Suppose we generate a periodic signal, as we did with the `wave` function. By introducing some noise, we can then analyze the model's performance with respect to this added noise. Specifically, if I generate a signal with a noise component of amplitude 0.5, ideally, we would expect the model's predictions to exhibit a maximum error of around 0.5.

6.1 Data generation

First of all, let's create a simplified version of the signals:

```
test_noise = 0.5
signal_1 = wave(15, resolution, 5) + test_noise
signal_2 = wave(5, resolution, 2, np.cos) + test_noise
signal_3 = wave(7, resolution, 3, np.cos) + test_noise
signal_4 = wave(6, resolution, 4) + test_noise
signal_5 = wave(10, resolution, 6) + test_noise


signals = [signal_1, signal_2, signal_3, signal_4, signal_5]
```



As we can observe, we have generated additional periodic signals with a noise level of three. Now, we can evaluate whether our model performs as expected (i.e., with an MSE ≤ 9) or if further adjustments are necessary to enhance its performance.

	Mean Squared Error				
	LSTM	***BiLSTM-30***	***BiLSTM-60***	***3-branch no attn***	***3-branch attn***
Signal 1	3.547	4.509	29.322	19.583	9.962
Signal 2	0.533	0.869	1.158	4.100	2.135
Signal 3	0.529	0.771	0.955	7.577	3.419
Signal 4	1.019	1.496	2.225	8.362	7.366
Signal 5	0.343	0.432	0.336	6.024	1.388

6.2 Comments on Results

The Mean Squared Error (MSE) values across the signals suggest that our models handle the introduced noise (amplitude 0.5) with varying levels of effectiveness. Given the expected target MSE of around 0.7 (due to noise amplitude), the models perform reasonably well in most cases, though certain signals reveal room for improvement.

For **Signal 1**, the LSTM model achieves an MSE of 3.547. While the model captures some characteristics of the periodic pattern, this error is still significantly higher than the expected 0.7, indicating a struggle to effectively handle the noise. The BiLSTM-30 and BiLSTM-60 models perform even worse, suggesting that added complexity may impact generalization for this signal.

On **Signal 2**, the LSTM model performs exceptionally well, achieving an MSE of 0.533, below the target MSE. This result suggests the model is highly effective at learning the underlying patterns of this periodic signal, even with the added noise. Notably, the 3-branch model with attention shows a relatively low MSE of 2.135, supporting the idea that attention mechanisms can enhance performance on certain signal types.

For **Signal 3**, the MSE values for both LSTM (0.529) and BiLSTM (0.771) are close to the expected 0.7 target, showing the models' strong capability to capture periodicity in this case. However, the 3-branch model without attention exhibits a much higher MSE (7.577), suggesting that attention mechanisms might indeed be beneficial for handling noise in more complex signals.

Signal 4 shows a less favorable result, with the LSTM model achieving an MSE of 1.019 and the BiLSTM models performing worse, indicating additional adjustments may be needed to improve performance on this signal. The increased error here suggests that further tuning may be necessary to enable these models to handle noise as effectively as anticipated.

For **Signal 5**, the LSTM model achieves an MSE of 0.343, which is well below the expected target of 0.7. This indicates strong performance in capturing the signal's periodic nature even with noise. The BiLSTM models also show favorable results (0.432 and 0.336), reinforcing the idea that the model architectures are well-suited for this type of signal. The 3-branch model, with an MSE of 6.024, however, suggests that this approach may need further refinement, as it does not perform as well as the simpler LSTM architecture in this instance.

We observed that the new version of the model, which incorporates more data, is significantly worsening our predictions. Could this be due to a lack of complexity in the signals?

Now, let's return to a more challenging task.

I'm gonna generate 10 different signals, each one with an high complexity in order to see if our model is capable to handle in a better way. After the training of the model the results obtained are:

	Mean Squared Error				
	Simple LSTM	***BiLSTM (seq=30)***	***BiLSTM (seq=60)***	***3 branch - no attention***	***3 branch - attention***
Signal 1	30.324	27.153	36.378	51.064	46.287
Signal 2	19.714	17.782	26.935	27.789	31.257
Signal 3	29.360	29.242	39.691	40.598	41.099
Signal 4	13.525	13.479	15.430	29.004	27.012
Signal 5	15.991	15.127	26.483	23.687	28.791
Signal 6	22.206	21.189	27.957	41.230	37.313
Signal 7	13.294	12.960	18.223	18.714	19.138
Signal 8	16.602	16.820	20.459	21.913	22.930
Signal 9	20.960	21.258	25.762	43.769	38.721
Signal 10	20.638	23.315	30.280	33.749	34.571

How is it possible to see our last model is not working properly, worsening a lot the predictions.

Conclusions

In this work, we explored the effectiveness of various models for time series prediction, with a specific focus on LSTM and BiLSTM architectures combined with Fourier transformations and attention mechanisms. The experimental results indicate that while BiLSTM + Fourier models demonstrated potential in capturing complex periodic signals, performance varied across different signals, highlighting the importance of tuning model parameters according to signal characteristic.

Our analysis also underscored the role of attention mechanisms. For certain signals, the inclusion of attention on the frequency domain contributed to lower MSE, while for others, it introduced additional complexity without substantial improvement. The application of dropout, especially in the latest model iteration, was instrumental in reducing overfitting, indicating a clear benefit for models trained on limited data.

Ultimately, while the baseline LSTM model remained the most reliable, further optimization should focus on enhancing the model's ability to process common signal components in the Fourier transformation. Future work may prioritize refining frequency selection and attention mechanisms to better capture shared patterns, potentially through adaptive filtering techniques that improve the handling of recurrent signal characteristics.