Design of LSTM Forecasting System for COVID-19 Cases Prediction in the Canary Islands

By Ginevra Iorio

Abstract

There exist many Long Short-Term Memory univariate models that can be implemented in Python and used for time series forecasting. These models have been implemented in a short program that is used to predict future Canary Islands' Hospitals COVID-19 data, based on previously studied and cleaned datasets.

Introduction

The goal of the project is designing a trustful Long Short-Term Memory Neural Network to be applied in a time series forecasting of future COVID-19 cases (such as new patients, deaths, recovered, etc.) in the hospitals of the Canary Islands. After the study of the state of the art and the analysis and preparation of the available datasets, the forecasting system can finally be implemented.

The number of recorded cases of every dataset is the only variable changing over time in all the available datasets, while the various observations are single and recorded sequentially over equal time increments (everyday). This suggests that the data should be treated as a *univariate time series*, using the suitable existing univariate models. Univariate time series models are a type of specifications in which one seeks to model and forecast certain variables using just information contained in their own past values and possible present and past values of an error term [1].

The exploitation of the accessible time series data is modeled by the project in a Python development environment. Various LSTM univariate models are implemented in order to provide a useful comparation and elect the one that best foresees future data. A walkthrough of all the steps followed to achieve the aforementioned system will be presented in this report.

Environment Setup

PyCharm is the Python Integrated Development Environment (IDE) chosen to write the Python code that models this project. The first step to be followed for its use is the setup of the environment, that involves the check, and eventually update, of the current python version and the installation of the various packages together with that of a package manager.

Python version

The version installed and used for this project is 3.8.9, which has been selected by default with the installation of the package manager and is compatible with all the packages to be used.

```
ginevra@MacBook-Air-2 LSTM_workspace % python -V
Python 3.8.9
```

Package Manager

Conda is the package manager that is used to install the packages needed for the project. It is a cross-platform package and environment manager that installs and maintains packages from the *Anaconda* repository and from the *Anaconda Cloud*. This is the most well-liked method for teaching and using Python in machine learning, data science, and scientific computing. It makes managing and deploying packages simpler, having a substantial collection of libraries and packages to be used. Anyone may contribute to its development because it is free and open source [2].

The installation of *Conda* has been made through *Miniconda*, a small, bootstrap version of *Anaconda* that only includes *Conda*, Python, the packages they depend on, and a small number of other useful packages, including *pip*, *zlib* and a few others [3].

Needed Packages

TensorFlow. The main package needed for a deep learning application like this one is TensorFlow, an open-source library that works with tensors, which are multi-dimensional arrays. The foundation of its operations are data flow graphs with nodes and edges, needed to define the flow of the tensors that are given as input [4].

The previously chosen package management wasn't *Conda* by chance, in fact, it allows to obtain a number of benefits. The installation is easier since *Conda* automatically installs all required and compatible dependencies while in contrast, *pip* would require to manually install all of them. Moreover, starting with version 1.9.0, the *Conda Tensorflow* packages use the Intel® Math Kernel Library for Deep Neural Networks (Intel® MKL-DNN) that, when dealing

with computationally intensive deep learning applications, significantly improves the performance, which could be more than 8x quicker as visible in the graph [5].

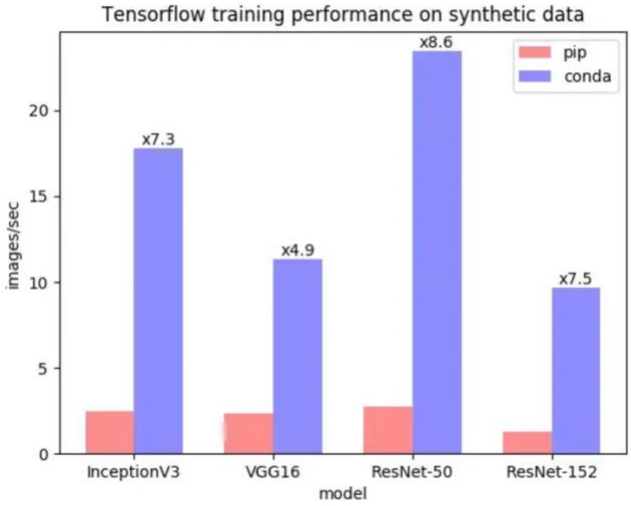

Chart taken from https://www.anaconda.com/tensorflow-in-anaconda/

The *TensorFlow*'s version utilized in this project is 2.10.0, which means that the performance improvement will be recorded thanks to the use of *Conda*.

```
ginevra@MacBook-Air-2 LSTM_workspace % conda list | grep tensorflow
tensorflow                2.10.0          cpu_py310h22f808f_0    conda-forge
```

Keras. Running on top of *TensorFlow*, *Keras* is a deep learning API that thanks to its simplicity, flexibility and scalability enables fast experimentation. Its core data structures are layers and models, between which the simplest and to-be-used one is the Sequential model, which is sufficient for a simple stack of layers, where each layer has precisely one input tensor and one output tensor [6]. This model is imported in the project inside the *univariate_models.py* program together with the layers that will be needed, which are shown in the following image.

```
import data_handler
from keras.models import Sequential
from keras.layers import Bidirectional
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import TimeDistributed
from keras.layers import ConvLSTM2D
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
```

The following table describes in detail which are the layers that have been used with a short definition and shows which LSTM models take advantage of them in the program:

Layer	Definition	LSTM models
Dense	A layer that is deeply connected with its preceding layer, meaning that the neurons of the layer are connected to every neuron of the preceding one. It is used to get the output in the format needed by the user, changing the dimensionality of the preceding layer.	Vanilla, Bidirectional, CNN, Stacked, Convolutional.
LSTM	This layer learns long-term relationships between time steps. It performs additive interactions, which might enhance gradient flow over lengthy periods during training.	Vanilla, Bidirectional, CNN, Stacked.
Bidirectional	This is a layer needed to connect two hidden layers of the opposite directions to the same output. This way the output layer gets the information from past and the future states simultaneously.	Bidirectional.
Flatten	Layer that flattens the input with no effect to the batch size	CNN, Convolutional.
TimeDistributed	It is a wrapper, that helps in adding a layer to each temporal slice of an input.	CNN.
ConvLSTM2D	It is comparable to an LSTM layer, but with a convolutional input and recurrent transformations.	Convolutional.
Conv1D	This layer generates a convolution kernel that is twisted with the layer input over a single spatial (or temporal) dimension, in order to generate a tensor of outputs.	CNN.
MaxPooling1D	By picking the highest value over a spatial window of the size pool size, it downsized the input representation.	CNN.

NumPy. It is Python's core scientific computing library. It offers various derived objects like masked arrays and matrices, a range of routines for quick operations on arrays and, most importantly, a multidimensional array object, *ndarray*, which contains n-dimensional arrays of homogeneous data types.

The *ndarray* is convenient in comparison to standard Python sequences because it has a fixed size at creation, all its elements have the same size in memory, it facilitates advanced mathematical operations on large data sets, and since many Python packages are using *NumPy* arrays, it is more straightforward to directly use them installing the package [7].

Pandas. This Python module offers quick, adaptable, and expressive data structures that are intended to make dealing with "relational" or "labeled" data simple and natural. It serves as Python's core, being a high-level building block used for performing useful real world data analysis. It aims to be the most robust and adaptable open-source data analysis/manipulation tool accessible in any language.

Its main features are simple handling of missing data, both implicit and explicit data alignment, size mutability, giving the opportunity to add or remove columns any time Additionally, it makes it simple to transform ragged, inconsistently indexed data from various *NumPy* and Python data structures into DataFrame objects. It also allows flexible datasets reshaping, subsetting and intuitive mergings, so it is the ideal tool to be used when working with data [8].

Data Preparation

All the data sets to be studied are situated inside the *csv_files* directory. After being retrieved with a simple *csv_read* function found in the *Pandas* library, each dataset is indexed by its date, transformed to datetime format.

```
# indexing the data with the Date column, after transforming it to datetime
df.index = pd.to_datetime(df['Date'], format='%d/%b/%Y')
```

The univariate sequence that will be serving as input of the model is then created by taking the entries of the 'NumCases' column of each dataset. There is, though, the need of a further preparation to be done before these series can be modeled.

The LSTM model will train a function that converts a series of previous observations from input to output, so these observations need to be split up into different instances, which will be needed by the LSTM in the learning process. Therefore, a *split_sequence* function is created inside the *data_handler.py*, the script designed to handle all the functions that will execute operations with the data. This function takes as input the previously defined sequence and an integer, *n_steps*, which defines the number of input samples that will be fed to the LSTM model for the prediction.

CNN and Convolutional LSTM models need *n_steps* to be an even number, since the samples obtained by the *split_sequence* function will later need to be reshaped, being split into two sub-samples of two time-steps each. Accordingly, for these two special cases *n_steps* has been chosen equal to 4. The rest of the model types have been set to have, on the other hand,

n_steps = 3, since a smaller number provides more samples for the LSTM input, which entails obtaining a more accurate result from the forecasting.
Let's consider the case where *n_steps* = 3: it means that the series is split into samples made up of three elements, that are then stored inside the array X. In the meanwhile, another array y containing the following values in the sequences of the samples is created and will be used as output for the prediction.

```
def split_sequence(sequence, n_steps):
    X, y = list(), list()
    i = 1

    while i < (len(sequence) - n_steps):
        # finding the end of this pattern
        end_pat = i + n_steps

        # gathering input and output parts of the pattern
        X.append(sequence[i:end_pat])
        y.append(sequence[end_pat])

        i = i + 1

    return array(X), array(y)
```

The obtained sequence of samples will be then split into three different datasets: the training, validation and testing sets.

<u>Training Data</u>

This is the set used for learning by Machine Learning algorithms, which generally are taught how to generate predictions or complete a specified task using this kind of datasets. One advantage of ML algorithms is that the accuracy of the model increases with training. Therefore, having a high volume of data for training is always preferable. In this project, the 80% of the dataset will be used for training.

Validation Data

It is used to evaluate the performance and accuracy of the model during the training phase. It helps avoiding overfitting and underfitting by preferring one model over the other, based on their performance on the test set. In the meanwhile, the model hyperparameters are adjusted, being updated with higher level ones. As a result, even if the model occasionally sees this data, it never learns from it, so the model is indirectly impacted by this set. 10% of the available data will represent the validation set in this project.

Testing Data

It is the subset of data that is used to impartially assess how well a final model fits the training dataset. Consequently, it is only used once a model has finished its training phase, when the hyperparameters can no longer be adjusted. The testing set will be chosen as the last 10% portion of the dataset [9].

Splitting the Data

For the sake of dividing the samples obtained after the *split_sequence* function of *univariate_models.py* into the three different sets, another function, *split_samples* is called inside the same script.

```
# splitting samples into training, validation and testing sets
X_train, y_train, X_val, y_val, X_test, y_test = data_handler.split_samples(X, y)
```

This function is defined inside the *data_handler.py* script and takes as input the two sequences of samplings, X and y, and divides them into training, validation and test sets.

```
# function that splits a sequence into train, validation and test sets
def split_samples(sequence1, sequence2):
    train_length, val_length = define_lengths(sequence1)
    X_train, y_train = sequence1[:train_length], sequence2[:train_length]
    X_val, y_val = sequence1[train_length:train_length+val_length], sequence2[train_length:train_length+val_length]
    X_test, y_test = sequence1[train_length+val_length:], sequence2[train_length+val_length:]

    return X_train, y_train, X_val, y_val, X_test, y_test
```

In order to define the lengths of the three sets, another function, *define_lengths*, needs to be called. Using the percentages listed above on the input sequence, the dimensions of the training and validation datasets are returned. The testing set's length doesn't need to be

defined as it will be made up by the remaining portion of the input sequence. Moreover, since X and y have the same length, this function only needs one of the two as input.

```
# function that defines the length in percentage of the train and validation sets
def define_lengths(seq):
    train_length = int(len(seq)*0.8)
    val_length = int(len(seq)*0.1)

    return train_length, val_length
```

LSTM Models

There exist multiple types of univariate LSTM models: Vanilla, Bidirectional, Stacked, CNN and Convolutional. In order to forecast the COVID data in the best way possible and have a comparison of the different models, they have all been implemented in the project and the user is given the possibility to choose which model to use at each run, after having chosen the dataset to study. All these models are defined inside the *univariate_models.py* program.

Vanilla LSTM

In a vanilla LSTM, the prediction is made using the output layer, which has only one hidden layer of LSTM units.

```
# ---------------- VANILLA LSTM MODEL ----------------
model = Sequential()
model.add(LSTM(units=32, activation='relu', input_shape=(n_steps, n_features)))
```

The LSTM *Keras* layer is added to the model with the specification of the following hyperparameters:

- *units*, which is the dimension of the hidden state, also known as output. It also determines the number of parameters in the LSTM layer. Having more units is beneficial since it enables the network to retain more complex patterns due to the higher dimension of hidden states;
- *activation,* that indicates which activation function to use. *ReLu* has been chosen since, in comparison to the *sigmoid* and *TanH*, it is the most advanced, totally eliminating limitations like the Vanishing Gradient Problem;
- *input_shape,* the shape of the input, or the amount of time steps and features the model requires as input for each sample.

Stacked LSTM

This model is made up by multiple hidden LSTM layers stacked one on top of the other. It is possible to use the 3D output from a hidden LSTM layer as input to a subsequent layer by using the *return sequences=True* option on the layer.

```
# ---------------- STACKED LSTM MODEL ----------------
model = Sequential()
model.add(LSTM(units=32, activation='relu', return_sequences=True, input_shape=(n_steps, n_features)))
model.add(LSTM(units=32, activation='relu'))
```

Bidirectional LSTM

BiLSTM can produce a more meaningful output, combining LSTM layers from both the past and the present: one takes the input in a forward direction and the other in a backwards direction. A Bidirectional LSTM may be implemented by enclosing the first hidden layer in a wrapper layer called Bidirectional.

```
# ---------------- BIDIRECTIONAL LSTM MODEL ----------------
model = Sequential()
model.add(Bidirectional(LSTM(units=32, activation='relu'), input_shape=(n_steps, n_features)))
```

CNN LSTM

Convolutional Neural Network (CNN) layers are used in the CNN LSTM architecture to extract features from input data, assisted by LSTMs in sequence prediction. This particular kind of neural network was created to be used use with two-dimensional picture inputs, but when features are automatically extracted and learned from one-dimensional sequence data, such as univariate time series data, it can also be quite successful.

```
# ---------------- CNN LSTM MODEL ----------------
model = Sequential()
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1, activation='relu'),
                          input_shape=(None, n_steps, n_features)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(units=32, activation='relu'))
```

By wrapping the CNN model in a TimeDistributed wrapper, the same model can be used when separately reading in each sub-sequence of the data. The first layer is a convolutional layer, which reads over the subsequence and requires some hyperparameters to be specified:

- *filters*, the times the input sequence is read or interpreted;
- *kernel_size,* which is the number of time steps contained in each "read" operation of the input sequence.

The CNN also needs to be made of a max pooling layer, which reduces the filter maps to half their size and incorporates the most important features. These structures are then reduced to a single one-dimensional vector thanks to the *Flatten* layer before being utilized as the LSTM layer's input.

ConvLSTM

It's another type of CNN LSTM, where each LSTM unity can perform a convolutional reading of the input. This model may be used for univariate time series forecasting even though it was designed for reading two-dimensional spatial-temporal data.

```
# ---------------- CONVLSTM MODEL ----------------
model = Sequential()
model.add(
    ConvLSTM2D(filters=64, kernel_size=(1, 2), activation='relu', input_shape=(n_seq, 1, n_steps, n_features)))
model.add(Flatten())
```

Regarding the quantity of filters and the size of the kernel, we may characterize the ConvLSTM as a single layer and since we are dealing with one-dimensional series, the kernel's row number is always fixed to 1. The model's output must then be flattened in order to be understood and a forecast to be made.

Output Layer

A Dense layer is finally added to every model, setting its units to 1 in order to have a one-dimension output.

```
model.add(Dense(units=1))
```

Building the Model

After the models and all their layers have been defined, the function *build_model* of the *data_handler.py* is called.

```
def build_model(model, X_train, y_train, X_val, y_val, X_test):
    # compiling model
    model.compile(optimizer='adam', loss='mse')
    # model.compile(loss=MeanSquaredError(), optimizer=Adam(learning_rate=0.0001), metrics=[RootMeanSquaredError()])

    # fitting model
    cp1 = ModelCheckpoint('model1/', save_best_only=True)
    model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, callbacks=[cp1])

    # modeling the testing dataset and printing the results
    model1 = load_model('model1/')
    test_predictions = model1.predict(X_test).flatten()

    return test_predictions
```

This function executes three different actions on the model:

1. Compile. This *Keras* function is used to define the *optimizer*, *loss* and the *metrics* that will be used by the training function.

   Optimizers are techniques that modify the weights and learning rates of the neural network in order to minimize losses. The *Adam* algorithm adjusts the learning rate for each weight in the neural network using estimates of the first and second moments of the gradient. It is chosen as it is the most effective stochastic optimization, needing just first-order gradients when memory is insufficient [10].

   The *loss function* analyzes how effectively the neural network represents the training data by comparing the target and the output values. The goal is to reduce this difference during training. The chosen loss function, MSE, calculates the average of the squared difference between the target and the predicted outputs. The loss will grow dramatically if a projected value is much larger than or less than its target value since this loss function is particularly sensitive to outliers [11].

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

   At this stage of the study, no *metrics* were used to evaluate the output, but they will be added later on.

2. Fit. Function called to train the model by slicing the data into *batches* and repeatedly iterating over the entire dataset for a given number of *epochs*. It determines how

effectively a machine learning model predicts data that is similar to that on which it was trained.

Its first three arguments are the input and target training datasets and the validation data previously defined as validation datasets, that will be used to check the performance of the model after each epoch of training.

The number of *epochs* is used to determine how many times the input data will be go through the whole training process, which should be an optimal number in order to avoid overfitting or underfitting.

The *callbacks* are a collection of operations used to receive a glimpse of the model's internal states and statistics. The *ModelCheckpoint* callback was used in order to save the model in a checkpoint file *cp1* at the end of every epoch if it's the best seen so far. The best model will be then used for prediction.

3. Predict. On the basis of the trained model, which is taken as input, this function predicts the labels of the data values. In order to map and forecast the labels for the test data, it works on top of the training model and using the previously learnt labels.

Finally, the test predictions that have just been made are returned by the *build_model* function.

Plotting the Predictions

The last action performed by every model is calling the *plot_results* function of the *data_handler.py*.

```
def plot_results(y_test, test_predictions, test_dates, title):

    # plotting the testing dataset in comparison with the actual dataset
    test_results = pd.DataFrame(data={'Test Predictions': test_predictions, 'Actuals': y_test})
    plt.plot(test_dates, test_results['Test Predictions'], label='Test Predictions')
    plt.plot(test_dates, test_results['Actuals'], label='Actual Values')
    plt.xlabel('Dates')
    plt.ylabel('Cases')
    plt.legend()
    plt.title(title)
    plt.show(block=True)
```

This function takes as input the target testing dataset, y_test, the test predictions returned by the *build_model* function, which will represent the y-axis, the title of the to-be-built graph

and the dates for which the testing dataset has been described, that will be displayed on the x-axis and are retrieved inside each model of *univariate_models.py* as:

```
# finding the dates for which the testing set is defined, to be used on the x axis of the graph
test_dates = df.index[len(X_train) + len(X_val):len(X_train) + len(X_val) + len(X_test)]
```

plot_results uses *Pandas DataFrame* structure to store and label the previously made predictions and the actual target values. The structure is then used to plot both these values on the y-axis of a graph, being the dates of the testing dataset chosen as x-axis. The legend for the graph and the title will also be shown.

MAIN Function

The main function is called inside the *init.py* script.
When the program is run, the list of all csv files containing the datasets to be studied is shown to the user in a numbered list, printed on the console by a for loop inside the *csv_files* directory.

```
# asking the user to choose the dataset to study
print("\n\nWhich csv file would you like to open?\n")
i = 1
for file in os.listdir(currDir + '/csv_files'):
    if file == '.DS_Store':
        continue
    print(str(i) + ' - ' + file)
    i = i+1

# asking for an input
print("\nInsert number: ")
file_num = input()
```

The user is then required to choose one file inputting its corresponding number of the numbered list. The check on the correctness of the input, which can only be an integer between 1 and 14, is then made by:

```
# checking for wrong inputs
if not file_num.isdigit():
    file_num = 0

while not (1 <= int(file_num) <= 14):
    print("Invalid number! Try again: ")
    file_num = input()

    if not file_num.isdigit():
        file_num = 0
```

Once the file to be opened has been chosen, it is loaded inside the variable *df*, while its name is loaded inside *dataset_name*, by calling the function *switch_file* of the *data_handler.py*. Based on the number inputted by the user, this function reads the correct csv file and returns it together with its defined title.

```
# function that, depending on the number chosen by the user, reads the right csv file
def switch_file(num, currDir):
    if num == "1":
        return pd.read_csv(currDir + '/csv_files/cases.csv'), 'COVID Cases'
    elif num == "2":
        return pd.read_csv(currDir + '/csv_files/discharged.csv'), 'Discharged Patients'
    elif num == "3":
        return pd.read_csv(currDir + '/csv_files/allCOVbeds.csv'), 'All Hospital COVID Beds'
    elif num == "4":
        return pd.read_csv(currDir + '/csv_files/accumulatedCases.csv'), 'Accumulated COVID Cases'
    elif num == "5":
        return pd.read_csv(currDir + '/csv_files/accumulatedDis.csv'), 'Accumulated Discharged Patients'
    elif num == "6":
        return pd.read_csv(currDir + '/csv_files/inCOVpatients.csv'), 'Incoming COVID Patients'
    elif num == "7":
        return pd.read_csv(currDir + '/csv_files/inTOTpatients.csv'), 'Total Incoming Patients'
    elif num == "8":
        return pd.read_csv(currDir + '/csv_files/resPatients.csv'), 'Respirator Patients'
    elif num == "9":
        return pd.read_csv(currDir + '/csv_files/deaths.csv'), 'COVID Deaths'
    elif num == "10":
        return pd.read_csv(currDir + '/csv_files/COVnoRes.csv'), 'COVID Patients Without Respirator'
    elif num == "11":
        return pd.read_csv(currDir + '/csv_files/COVwithRes.csv'), 'COVID Patients With Respirator'
    elif num == "12":
        return pd.read_csv(currDir + '/csv_files/allBeds.csv'), 'All Hospital Beds'
    elif num == "13":
        return pd.read_csv(currDir + '/csv_files/noResPatients.csv'), 'Patients Without Respirator'
    elif num == "14":
        return pd.read_csv(currDir + '/csv_files/accumulatedDeaths.csv'), 'Accumulated COVID Deaths'
```

The *init.py* continues its execution by asking the user which LSTM univariate model to use and presenting a numbered list of the implemented models: Vanilla LSTM, Stacked LSTM, Bidirectional LSTM, CNN LSTM and ConvLSTM.

```
# asking the user to choose one of the LSTM univariate models
print("\nWhich LSTM univariate model would you like to use?\n")
print("1 - Vanilla LSTM\n2 - Stacked LSTM\n3 - Bidirectional LSTM\n4 - CNN LSTM\n5 - ConvLSTM\n")
print("Insert number: ")
model_num = input()
```

The correctness of the input, which is required to be an integer between 1 and 5, is checked again by:

```
# checking for wrong inputs
if not model_num.isdigit():
    model_num = 0

while not (1 <= int(model_num) <= 5):
    print("Invalid number! Try again: ")
    model_num = input()

    if not model_num.isdigit():
        model_num = 0
```

Ultimately, the function *switch_model* of the *data_handler.py* is called inside the *main* function. Based on the last number inputted by the user, it calls the corresponding model function of the *univariate_models.py,* with the file and dataset name as parameters.

```
# function that, depending on the number chosen by the user, calls the right function
def switch_model(num, df, dataset_name):
    if num == "1":
        return univariate_models.vanilla_LSTM(df, dataset_name)
    elif num == "2":
        return univariate_models.stacked_LSTM(df, dataset_name)
    elif num == "3":
        return univariate_models.bidirectional_LSTM(df, dataset_name)
    elif num == "4":
        return univariate_models.CNN_LSTM(df, dataset_name)
    elif num == "5":
        return univariate_models.conv_LSTM(df, dataset_name)
```

This function will execute the previously mentioned data preparation, apply the correct LSTM model to the dataset and train it, obtaining the required prediction of the testing dataset, which will be shown in a plot together with the actual testing dataset values.

Conclusions

Time series observations about COVID-19 cases recorded in the hospitals of Canary Islands can be used for univariate forecasting, that implies they are the only variable to consider. Different types of univariate LSTM models have been implemented, with weights and hyperparameters set to default or to the optimal case possible. For this reason, during the next phase of the project, the experimentation, they will be tested and eventually changed, in order to obtain a predicted model that best fits the actual values and avoids overfitting and underfitting the curve.

Bibliography

[1] Brooks, Chris. “Univariate Time Series Modelling and Forecasting (Chapter 5) Introductory Econometrics for Finance.” *Cambridge Core*, Cambridge University Press, www.cambridge.org/core/books/abs/introductory-econometrics-for-finance/univariate-time-series-modelling-and-forecasting/F20C6744BB368689ED4B6C5333BA773F

[2] Ellis, Danielle. “What Is Anaconda for Python & Why Should You Learn It?” *HubSpot Blog*, HubSpot, https://blog.hubspot.com/website/anaconda-python

[3] “Miniconda.” *Miniconda - Conda Documentation*, https://docs.conda.io/en/latest/miniconda.html

[4] Banoula, Mayank. “What Is Tensorflow? Deep Learning Libraries and Program Elements Explained.” *Simplilearn.com*, Simplilearn, www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-tensorflow

[5] Gunjal, Siddhesh. “Why Conda Install Instead of Pip Install?” *Medium*, Analytics Vidhya, https://medium.com/analytics-vidhya/why-conda-install-instead-of-pip-install-ba4c6826a0ae

[6] Team, Keras. “Keras Documentation: About Keras.” *Keras*, https://keras.io/about/

[7] “What Is Numpy?” *What Is NumPy? - NumPy v1.24 Manual*, https://numpy.org/doc/stable/user/whatisnumpy.html

[8] “Package Overview.” *Package Overview - Pandas 1.5.3 Documentation*, https://pandas.pydata.org/docs/getting_started/overview.html

[9] Shah, Tarang. “About Train, Validation and Test Sets in Machine Learning.” *Medium*, Towards Data Science, https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

[10] “Adam.” *Adam - Cornell University Computational Optimization Open Textbook - Optimization Wiki*, http://optimization.cbe.cornell.edu/index.php?title=Adam

[11] Yathish, Vishal. “Loss Functions and Their Use in Neural Networks.” *Medium*, Towards Data Science, https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9