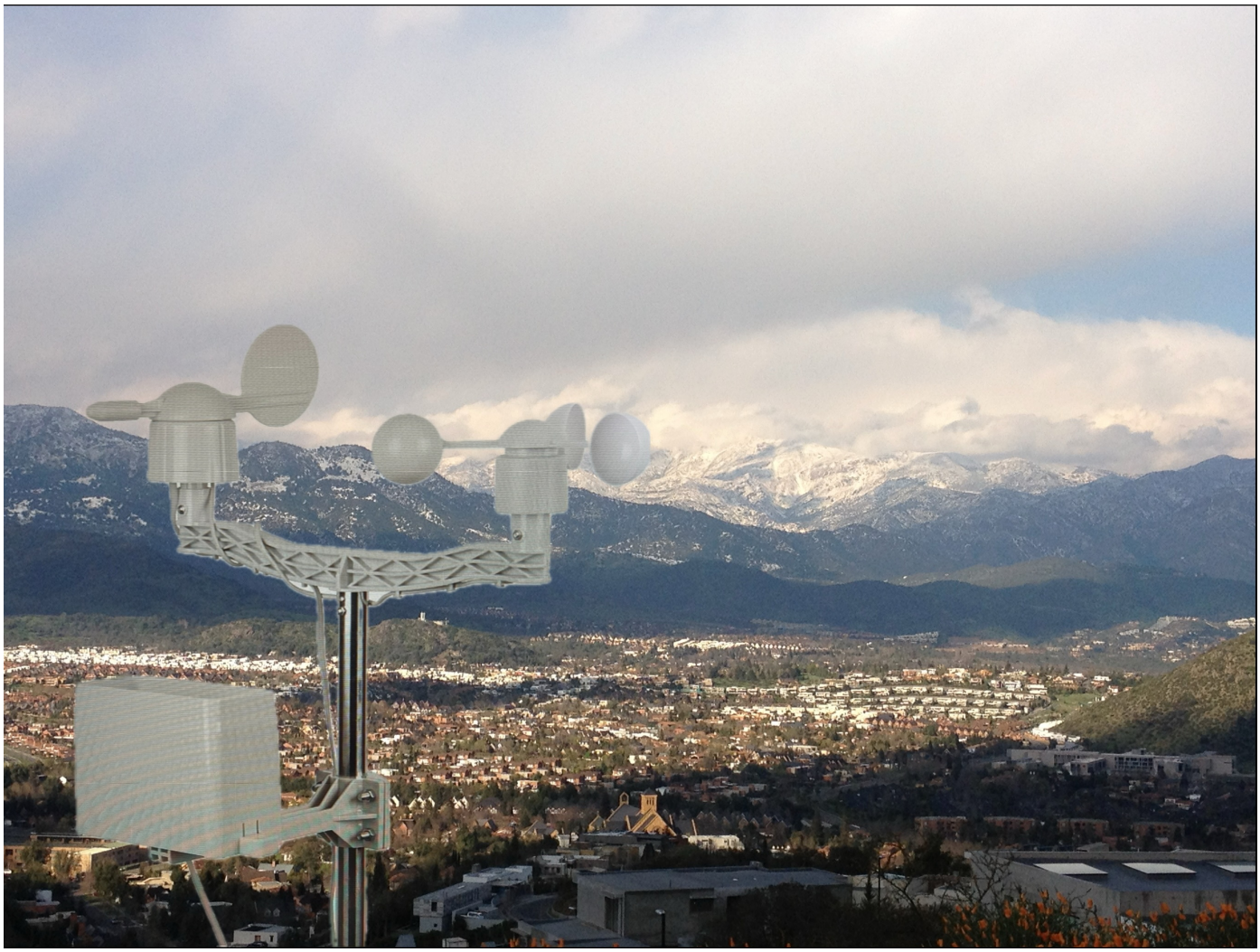
Temperature Prediction using a TinyML LSTM model

The main idea of this project is to use historical weather data of a specific location to train a model to predict its future temperature. The trained LSTM (Long Short-Term Memory) model will be converted with TensorFlow Lite to be deployed on a microcontroller. The prediction will be tested using an XIAO ESP32S3.



'

Lo Barnechea, Chile

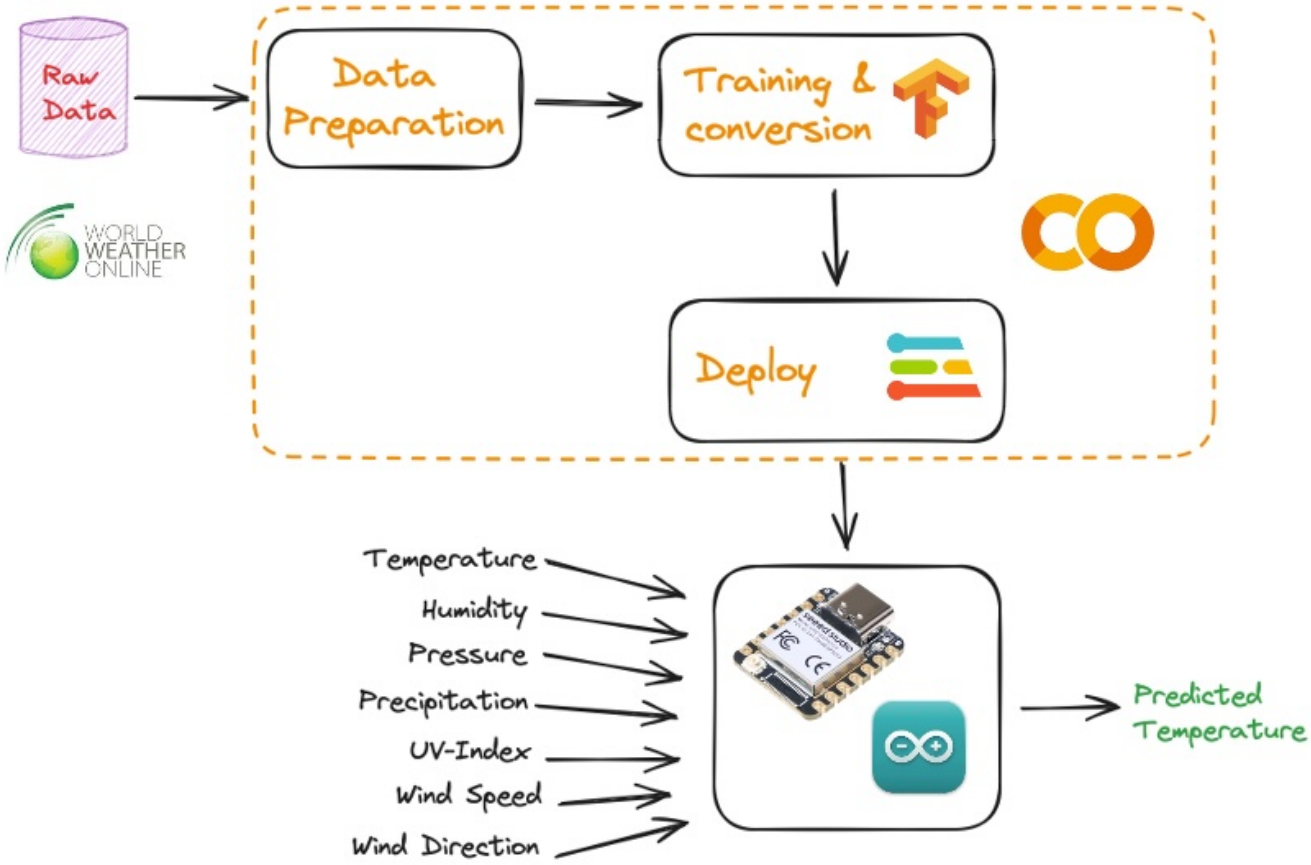
Introduction

LSTM (Long Short-Term Memory) networks are particularly well-suited for temperature prediction tasks, where the goal is to forecast future values based on past observations of various parameters such as temperature, humidity, pressure, precipitation, UV index, wind speed, and wind direction. The strength of LSTMs lies in their ability to capture temporal dependencies and patterns over time, which are intrinsic to weather data.

Unlike traditional time-series forecasting models that struggle with long-term dependencies or complex nonlinear relationships, LSTMs can learn to recognize and propagate important information through the hidden state over many time steps. This capability allows them to effectively remember and utilize past conditions when predicting future weather patterns.

Marcelo Rovai @ February, 2024

For training, 14 years of weather data from Lo Barnechea City in Chile will be used. The data will be verified, cleaned, transformed, and normalized using Python on a Google Colab Notebook. The training will be done using TensorFlow, with the final model converted to TensorFlow Lite and deployed to a microprocessor using the Edge Impulse Python SDK (also on CoLab).



The Dataset

We will use historical data from the World Weather Online website. To do so, we will need an API Key. To get one, we should sign up in the Developers area, where we can have a 30-day free trial period.

We will retrieve **hourly** (*frequency*) historical data for the *city* of **Lo Barnechea** in Chile from **January 01, 2010** (start_date) until the last **March 25th** (*end_date*). So, let us define our variables:

```
api_key = 'YOUR KEY HERE'
city = 'Lo Barnechea'
start_date = datetime.strptime('2010-01-01', '%Y-%m-%d')
end_date = datetime.strptime('2024-03-25', '%Y-%m-%d')
frequency = '1'  # Hourly
```

> The **frequency** should be an *integer*: 1, 3, 6, 12, 24, where 1 hourly, 3 hourly, 6 hourly, 12 hourly (day/night), or 24 hourly (day average)

We will get the data in a JSON format (30 days for each file).

> The JSON file contains a lot of information. To learn about its content, go to https://www.worldweatheronline.com/weather-api/premium-api-explorer.aspx and explore the interactive API explorer.

To download the data, let's use a function to generate monthly date ranges:

```
def generate_month_ranges(start, end):
    current = start
    while current <= end:
        month_end = (current.replace(day=28) + timedelta(days=4)).replace(day=1) -
timedelta(days=1)
        if month_end > end:
            month_end = end
        yield current, month_end
        current = month_end + timedelta(days=1)
```

On your directory, create a folder named `data`. Now, we should loop through each month and download data:

```
for start, end in generate_month_ranges(start_date, end_date):
    encoded_city = quote(city)
    url = f'http://api.worldweatheronline.com/premium/v1/past-weather.ashx?key=
{api_key}&q={encoded_city}&format=json&date={start.strftime("%Y-%m-%d")}&enddate=
{end.strftime("%Y-%m-%d")}&tp={frequency}'

    response = requests.get(url)
    monthly_data = response.json()

    # Save the JSON for the month
    with open(f'./data/weather_{city}_{start.strftime("%Y_%m")}.json', 'w') as f:
        json.dump(monthly_data, f)
```

You should find several .json files in the data folder, such as *weather_Lo Barnechea_2010_01.json*, *weather_Lo Barnechea_2010_02.json*, and *weather_Lo Barnechea_2010_03.json*.

We will need to combine all those monthly files into a single one. Start specifying the directory where your JSON files are stored and create a list of all JSON files to concatenate:

```
directory = './data/'
file_pattern = 'weather_*_*.json'
json_files = glob(os.path.join(directory, file_pattern))
```

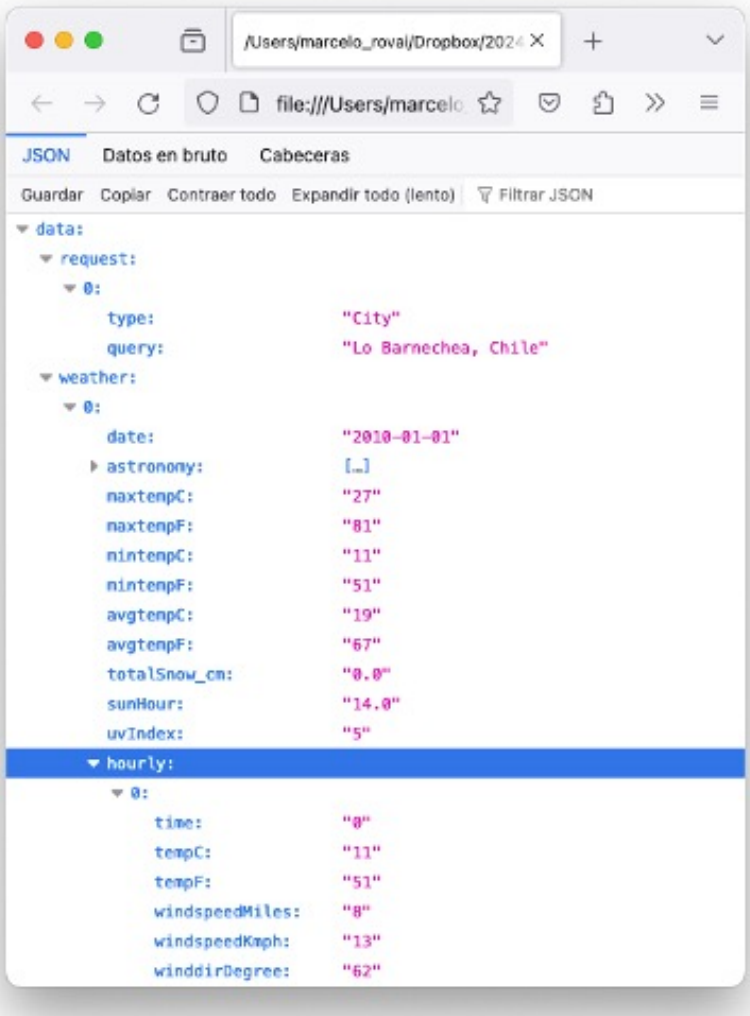
Initialize a list to store all weather data, looping through the files and reading the data:

```
all_weather_data = []
for filename in json_files:
    with open(filename, 'r') as file:
        data = json.load(file)
        try:
            # Append the 'weather' part of each file's data to the all_weather_data list
            all_weather_data.extend(data['data']['weather'])
        except KeyError as e:
            print(f"KeyError for {filename}: {e}")
```

Now that `all_weather_data` contains the concatenated weather data from all files, save the combined data to a new JSON file:

```
output_filename = './data/combined_weather_data.json'
with open(output_filename, 'w') as outfile:
    combined_data = {'data': {'weather': all_weather_data}}
    json.dump(combined_data, outfile, indent=4)
```

When you open the JSON file, you will find the bellow structure, with the top-level key 'data' containing the keys `'request'` and `'weather'`. Under the `weather`, we will find where our target data is stored `hourly`.



We will select the following numerical data:

- 'tempC',
- 'humidity',
- 'pressure',
- 'precipMM',
- 'uvIndex',
- 'windspeedKmph', and

- 'winddirDegree'

We will also include non-numerical data ('weatherDesc') describing the weather as Clear, Sunny, Rain, etc. We will not use this information to train the model, but it could help analyze the dataset.

From the combined JSON file, we will create a Pandas data frame:

```
# Load the combined JSON data
with open('../data/combined_weather_data.json', 'r') as file:
    combined_data = json.load(file)

# Extract the hourly data
hourly_data_list = []

for weather_day in combined_data['data']['weather']:
    date = weather_day['date']
    for hourly_data in weather_day['hourly']:
        # Flatten the hourly data and add the date to each entry
        hourly_data_flattened = {
            'date_time': f"{date} {int(hourly_data['time'])//100:02d}:00",  # Combines
date and time
            'tempC': hourly_data['tempC'],
            'humidity': hourly_data['humidity'],
            'pressure': hourly_data['pressure'],
            'precipMM': hourly_data['precipMM'],
            'uvIndex': hourly_data['uvIndex'],
            'windspeedKmph': hourly_data['windspeedKmph'],
            'winddirDegree': hourly_data['winddirDegree'],
            'weatherDesc': hourly_data['weatherDesc'][0]['value'],  # Assumes first
description is primary
        }
        hourly_data_list.append(hourly_data_flattened)

# Create a pandas DataFrame from the list of dictionaries
hourly_df = pd.DataFrame(hourly_data_list)
```

The data retrieved is not numerical (Int or Float), so we should convert all numerical columns. We will use `pd.to_numeric()` to convert temperature and other specified columns to a numeric type. The `errors='coerce'` argument ensures that if non-numeric values are encountered, they will be set to NaN ('Not a Number') instead of raising an error.

```
# Convert columns to numeric
numeric_cols = ['tempC', 'humidity', 'pressure', 'precipMM', 'uvIndex', 'windspeedKmph',
'winddirDegree']
hourly_df[numeric_cols] = hourly_df[numeric_cols].apply(pd.to_numeric, errors='coerce')
```

We should convert the 'date_time' column to datetime format, setting it as the index of the DataFrame. Do not forget also to sort the DataFrame by date_time:

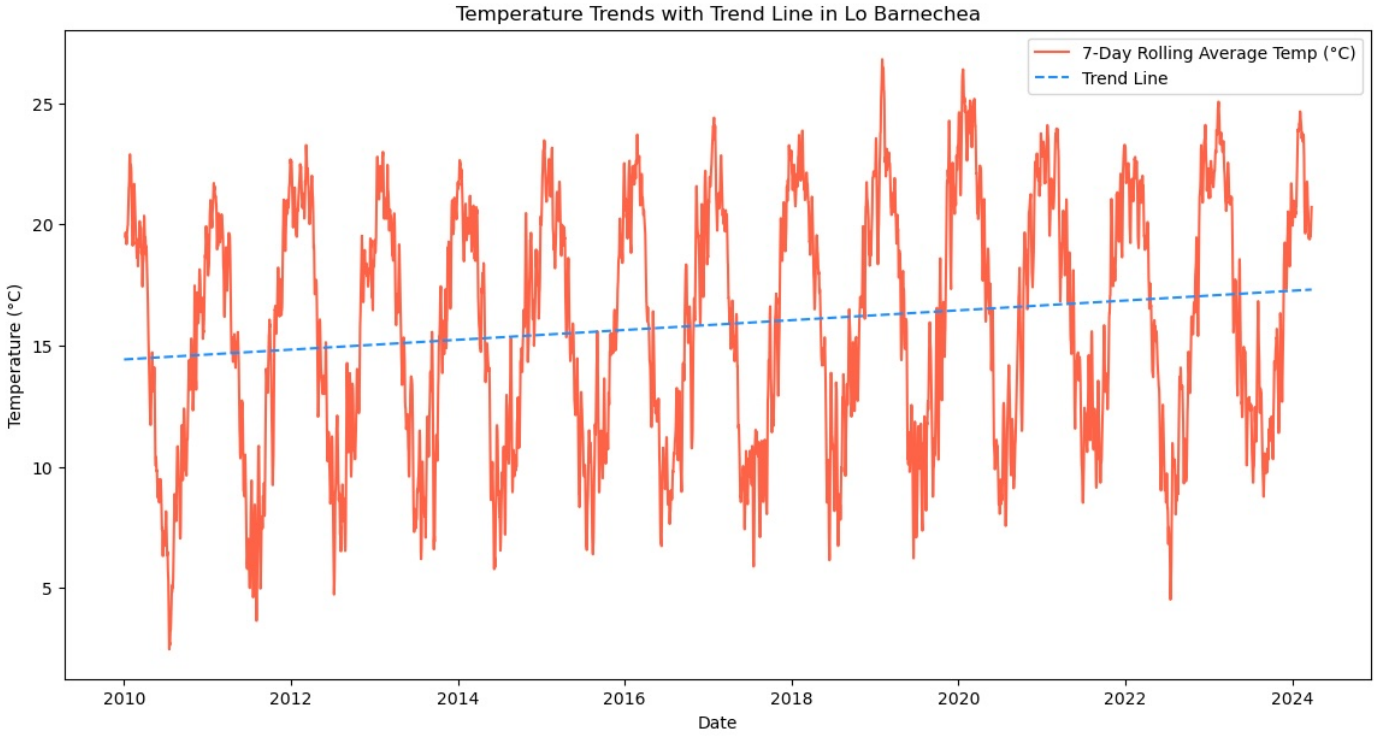
```
hourly_df['date_time'] = pd.to_datetime(hourly_df['date_time'], format='%Y-%m-%d %H:%M')
hourly_df.set_index('date_time', inplace=True)
hourly_df.sort_values(by='date_time', inplace=True)
```

Save the DataFrame to a CSV file:

```
csv_file_path = './data/weather_data_lo_barnechea_hourly.csv'
hourly_df.to_csv(csv_file_path, index=True)
```

> This CSV file will be used in a specific notebook to train our LSTM model.

You can explore the dataset with plots, providing valuable insights into the trends and patterns within your weather data. You can use basic visualizations using `pandas` and `matplotlib`, two powerful libraries for data analysis and visualization in Python. In notebook `01-Get_and_Explore_Data.ipynb`, you find some plots and analyses, such as the Temperature Trend for Lo Barnechea, Chile.



> The temperature has been increasing for over a decade, proving that we should care for our environment.

Preparing the data for Training

We should elect the relevant columns for our projects, which will be the numerical ones:

```
numeric_cols = ['tempC', 'humidity', 'pressure', 'precipMM', 'uvIndex', 'windspeedKmph',
'winddirDegree']
data_selected = hourly_df[numeric_cols]
data_selected.shape
```

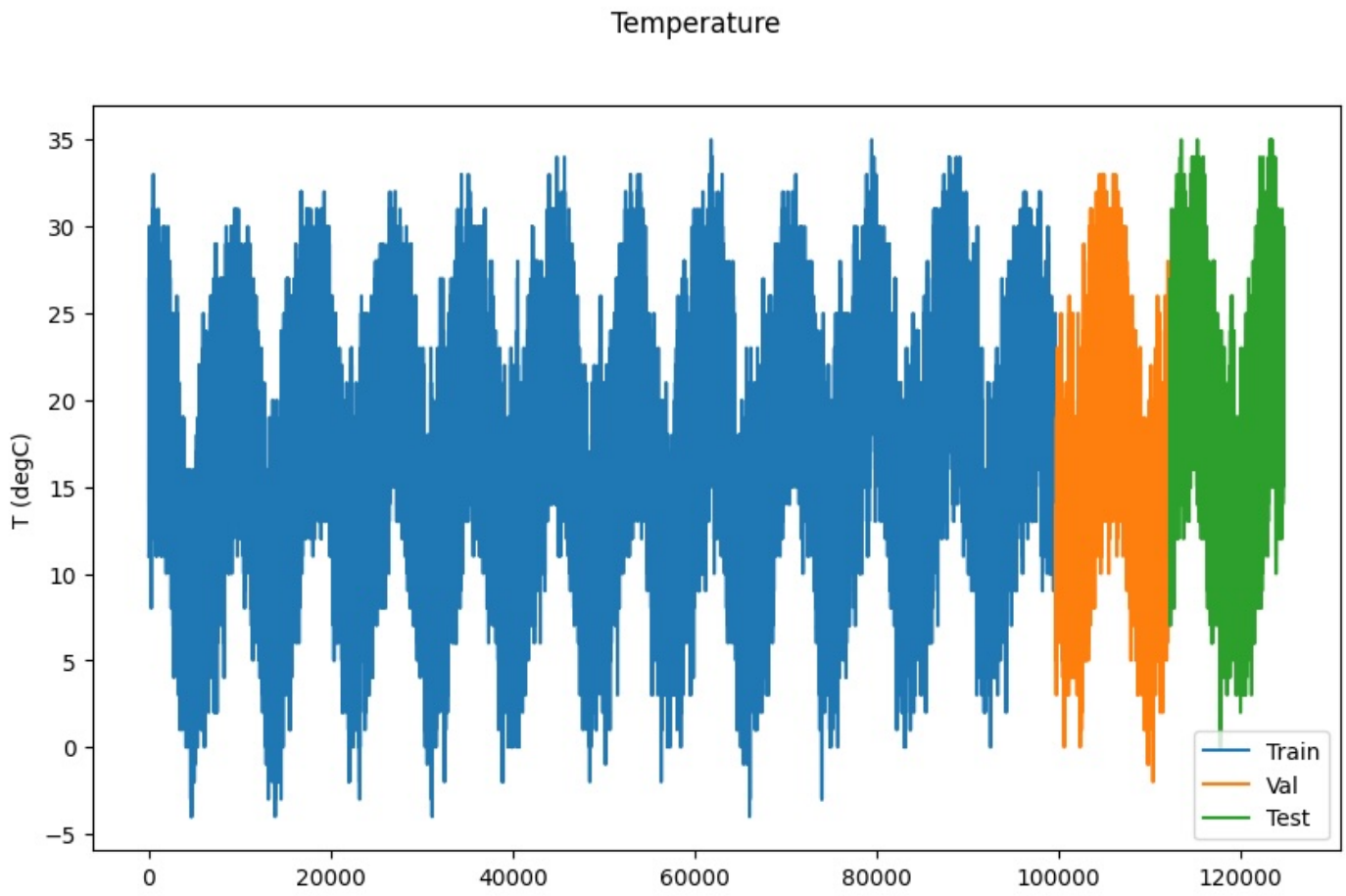
The dataset has the shape of (124752, 7).

We should split the dataset into training, validation, and testing sets, with 80% for training, 10% for testing, and 10% for validation.

```
SPLIT = 0.8

train_size = int(len(data_selected) * SPLIT)
val_size = int(len(data_selected) * (1-SPLIT)//2)
test_size = len(data_selected) - train_size - val_size

data_train = data_selected[:train_size]
data_val = data_selected[train_size:train_size + val_size]
data_test = data_selected[-test_size:]
```

We can plot those data for checking:

```
data_train['tempC'].plot(legend=True)
data_val['tempC'].plot(legend=True)
data_test['tempC'].plot(legend=True)
plt.legend(['Train', 'Val', 'Test']);
plt.suptitle('Temperature')
plt.ylabel('T (degC)');
```



The next step is to normalize the data so all features (columns) will range from 0 to 1:

```
scaler = MinMaxScaler(feature_range=(0, 1))
data_train_normalized = scaler.fit_transform(data_train)
data_val_normalized = scaler.transform(data_val)
data_test_normalized = scaler.transform(data_test)
```

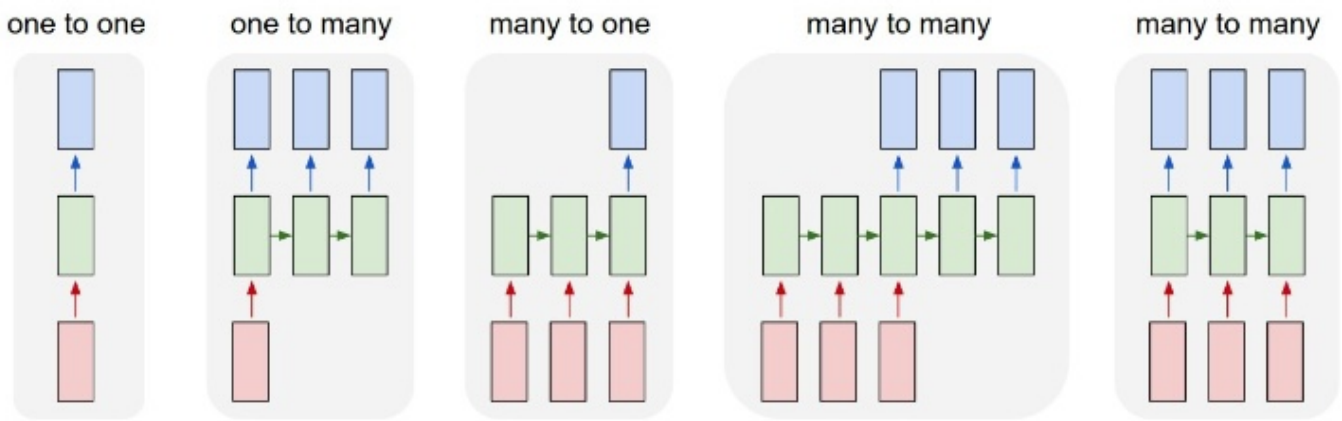
> We should save the scaler parameters to a text file for use during the inference.

Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNN)

LSTMs are designed to handle multivariate time series data, making them a fitting choice for incorporating multiple weather parameters to create a comprehensive predictive model. By learning from the intricate interactions among these parameters, an LSTM can make more accurate predictions that account for the combined effects of all contributing factors in the atmospheric system.

So, incorporating LSTM (Long Short-Term Memory) networks within TinyML (Tiny Machine Learning) applications can be quite powerful, especially for our project, which requires the analysis of sequential data directly on a microcontroller, as in our case, where we will implement the model on an XIAO ESP32S3.

LSTM models can be configured for different types of tasks as shown in [LSTM RNN in Keras: Examples of One-to-Many, Many-to-One & Many-to-Many](#):



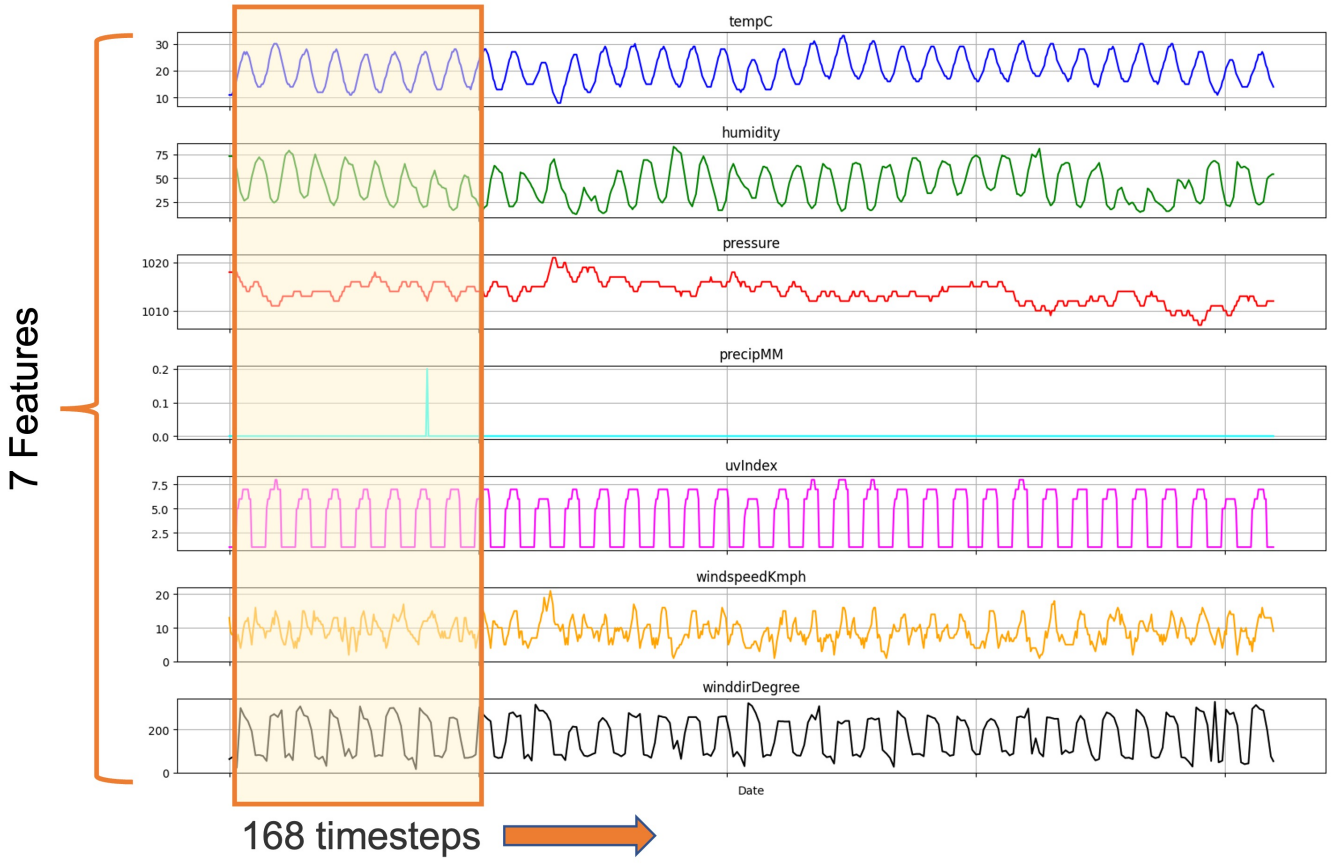
1. **One-to-One**: For tasks where there is a single input and a single output, often used for standard classification or regression tasks.
2. **One-to-Many**: For tasks that require one input to generate a sequence of outputs, like image captioning, where an image input generates a sequence of words as output.
3. **Many-to-One**: This is used for tasks where a sequence of data points leads to a single output. It is particularly suitable for sentiment analysis or, as in your case, weather forecasting, where a series of observations over time leads to a single forecasted value. The "many-to-one" structure enables the model to look at a sequence of data points, process the temporal relationships within that sequence, and condense all that information into a single predictive output, which would be the predicted temperature at a future time point.
4. **Many-to-Many**: For tasks where an input sequence maps to an output sequence, which can be synchronized or with different lengths, such as machine translation or speech recognition.

In our project (temperature prediction) using a set of weather-related parameters, a "**many-to-one**" LSTM is our natural choice. We are interested in predicting a single future temperature value from a sequence of past observations. The model needs to understand and remember patterns across the input sequence (such as daily or seasonal cycles) to predict the next step accurately. This sequence-to-value prediction is precisely what "many-to-one" LSTMs excel at.

Data Preparation for LSTM Forecasting

When preparing data for an LSTM model, a crucial step is to transform the time series dataset into sequences that the model can use to learn the patterns. Since LSTMs are designed to work with data sequences, we need to reformat our dataset into an input-output structure where the input is a sequence of data points leading up to a certain time, and the output is the value at the next step.

In this case, we will use one week of data by defining a **window of 168 (24 x 7) timesteps** to predict the temperature **one hour in the future**. This means that we use the data from the past 168 hours (which equates to one week of data) to predict the temperature for the next hour.



> Note that you can define future predictions, such as 3 or 6 hours. LSTMs can be more reliable for short-term forecasting. The further you try to predict the future, the more uncertain the predictions become.

Creating sequences for LSTM (Features)

We should create sequences of 168 timesteps with 7 features (Temp, Hum, etc). Let's define the variables:

1. **n_steps**: This variable defines the length of the input sequence for the LSTM. With `n_steps = 168`, each input sequence will consist of 168 consecutive hours of data.

2. **n_inputs**: We calculate the number of features in our dataset, corresponding to the number of different parameters we have for each timestep.

```
n_steps = 168
n_inputs = len(data_selected.columns)
```

We should define a function that creates the sequences the LSTM will use during training.

```
def create_sequences(data, n_steps):
    X, y = [], []
    for i in range(len(data) - n_steps):
        X.append(data[i:i + n_steps, :])  # All features
        y.append(data[i + n_steps, 0])    # Temperature only
    return np.array(X), np.array(y)
```

> It's important to note that the temperature is assumed to be the first column in the dataset, which is why we use index 0 when referencing the output data.

By the end of this process, we have two NumPy arrays:

- `X`: An array of input sequences, each with 168 timesteps and `n_inputs` features.
- `y`: An array of output temperatures, each corresponding to the temperature immediately following each input sequence.

This structured data is then used to train the LSTM model, with the network learning to predict the temperature at time `t` based on the data from the previous 168 hours.

Let's apply the function to our split dataset:

```
X_train, y_train = create_sequences(data_train_normalized, n_steps)
X_val, y_val = create_sequences(data_val_normalized, n_steps)
X_test, y_test = create_sequences(data_test_normalized, n_steps)
```

Design and Train the LSTM Model

Model Design

The LSTM architecture uses the Sequential API from Keras to implement a "many-to-one" type model. As described previously, this configuration is characterized by taking a sequence of data points as input and producing a single output value.

```
model = Sequential([
    LSTM(128,
         input_shape=(n_steps, X_train.shape[2])),
    Dense(1)
])
```

Here's a breakdown of the model's architecture:

1. **LSTM Layer**: The first layer in our model is an LSTM layer with 128 units (you can try another hyperparameter here). The `input_shape` parameter indicates that your model expects input data to be in a sequence (`n_steps`) of a certain number of features (`X_train.shape[2]`). Each sequence is fed into the model and processed by the LSTM layer, where temporal dependencies within the sequence are learned.
2. **Dense Layer**: After the LSTM layer, we have a Dense layer with a single neuron. This layer serves as the output layer and produces a single continuous value as the output — in our case, likely a temperature value. This setup is commonly used for regression tasks, such as forecasting a numeric value.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 128)               69632

 dense (Dense)               (None, 1)                 129

=================================================================
Total params: 69761 (272.50 KB)
Trainable params: 69761 (272.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

> The model has a total of 69,761 parameters, which means that, in terms of memory, we will have a cost of around 273 KB (4 bytes per parameter).

Model Compile

Optimizer: 'adam'

The optimizer is responsible for updating the neurons' weights in our network. 'Adam' stands for Adaptive Moment Estimation, and it is one of the most commonly used optimization algorithms because it combines the best properties of the AdaGrad and RMSProp algorithms to handle sparse gradients on noisy problems. Adam is efficient in computation and requires little memory, making it suitable for many neural network problems, including time series forecasting with LSTMs.

Loss Function: 'mse'

The loss function, 'mse', stands for Mean Squared Error. It measures the average squared difference between the actual and predicted values. In the context of our temperature forecasting model, it quantifies how close the model's predicted temperatures are to the actual temperatures from the dataset. Minimizing this value during training improves the accuracy of your model's predictions.

```
model.compile(optimizer='adam', loss='mse')
```

Model Training

After compiling the model, we must fit it to our training data using `model.fit()`, providing input sequences, corresponding target temperatures, and other training parameters like the number of epochs and batch size. This process iteratively adjusts the model weights to minimize the loss, ideally resulting in a model that can accurately predict future temperatures based on the provided sequence of input features.

But, before, let's define an "Early Stopping" callback using Keras. Early Stopping is a form of regularization used to avoid overfitting when training a machine learning model, especially with neural networks. It monitors the model's performance on a validation dataset and stops the training process if it stops improving.

```
early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    mode='min',
    restore_best_weights=True)
```

Here are the parameters used in your `EarlyStopping` callback:

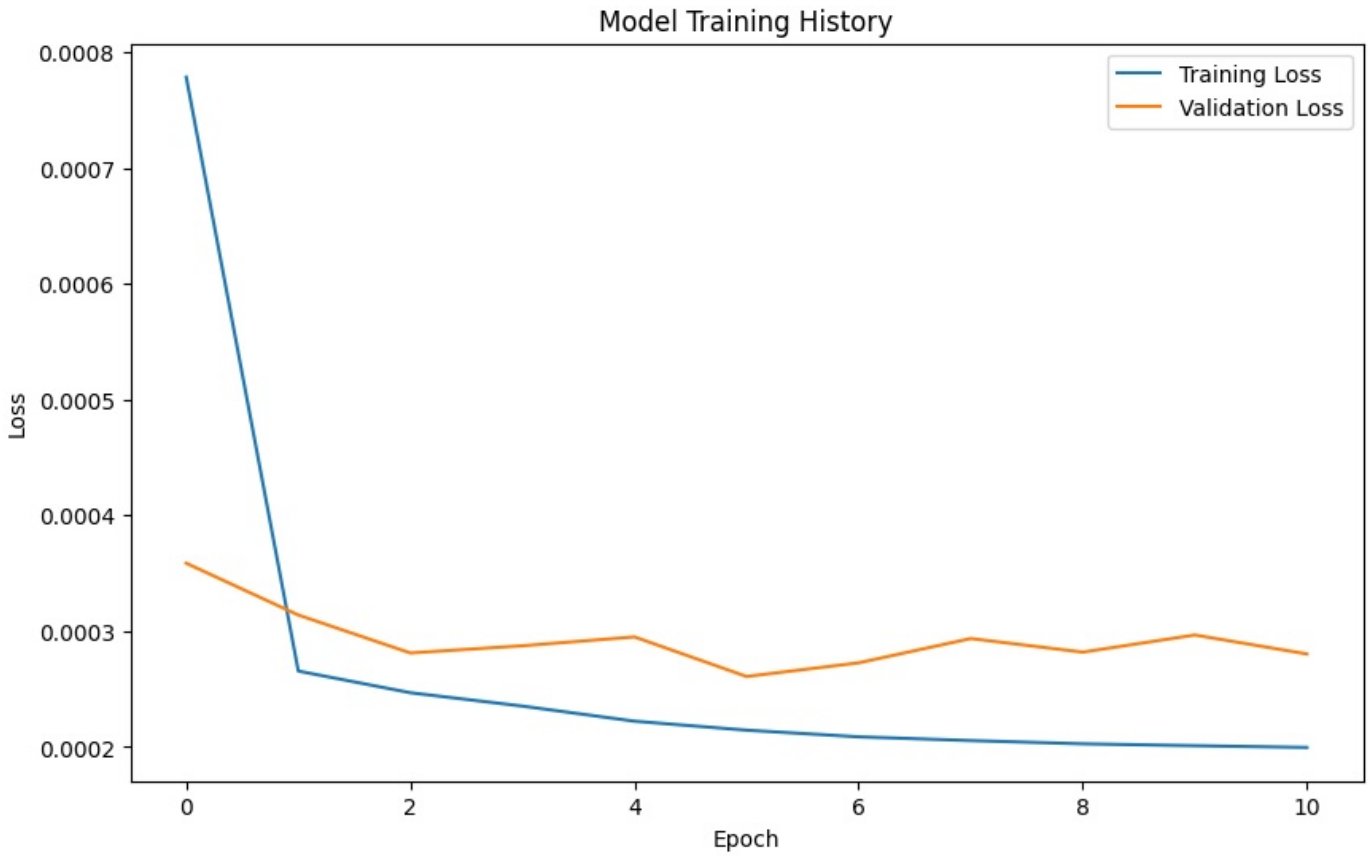
- **monitor**: This parameter specifies the metric to be monitored, which is `'val_loss'` in our case. The `'val_loss'` refers to the model's loss on the validation dataset. Monitoring this metric allows the callback to monitor the model's performance on unseen data.
- **patience**: This parameter defines the number of epochs with no improvement, after which training will be stopped. Setting patience=5 means that if the value of 'val_loss' does not decrease for 5 consecutive epochs, the training process will be halted. This gives the model some leeway to overcome small hiccups in training progress.
- **mode**: The `'min'` mode means that training will stop when the quantity monitored (`'val_loss'`) stops decreasing. This makes sense because less is better regarding loss; we want to minimize it.
- **restore_best_weights**: When set to `True`, this option restores model weights from the epoch with the best value of the monitored metric (`'val_loss'` in this case). This means that we will keep the model's best state even if the model's performance degrades in the epochs following the best one (within the patience period).

To utilize the `early_stopping` callback, we pass it to the `callbacks` parameter of the `model.fit()` method during training:

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=20,
    batch_size=32,
    callbacks=[early_stopping]
)
```

Our model will train on the training data (`X_train`, `y_train`), while also evaluating on a separate validation set (`X_val`, `y_val`). If the validation loss does not improve for five consecutive epochs, the training will stop early, and the model's weights will revert to those from the epoch with the lowest validation loss, effectively preventing overfitting and saving computational resources.

The training process was stopped on the 11th Epoch.



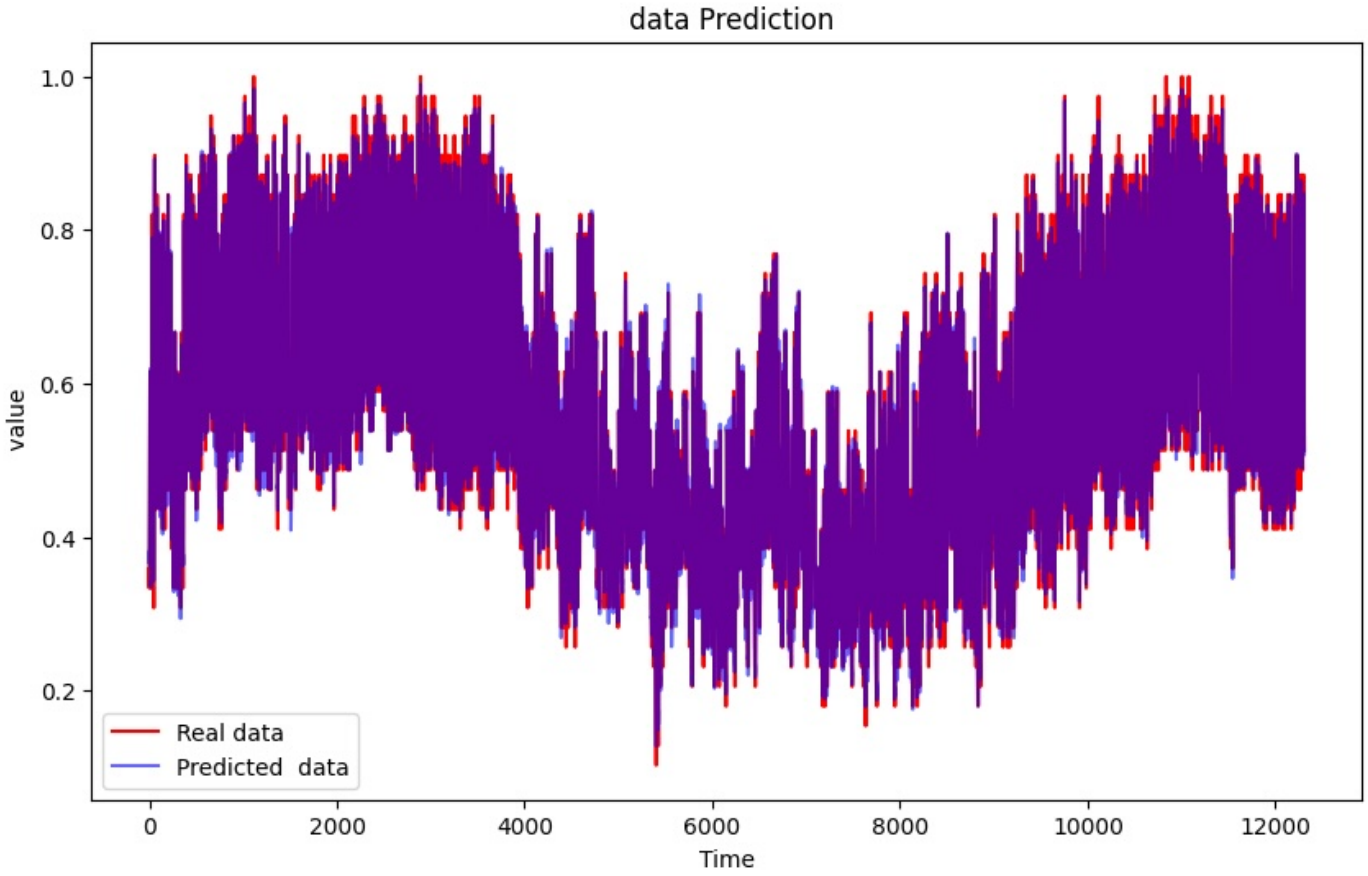
Model evaluation computed with RMSE (Root Mean Square Error), give us 0.021 on normalized data what is promising, indicating that our LSTM model performs well in forecasting temperatures based on the given features. Remember that our maximum value here is 1.

Data Prediction

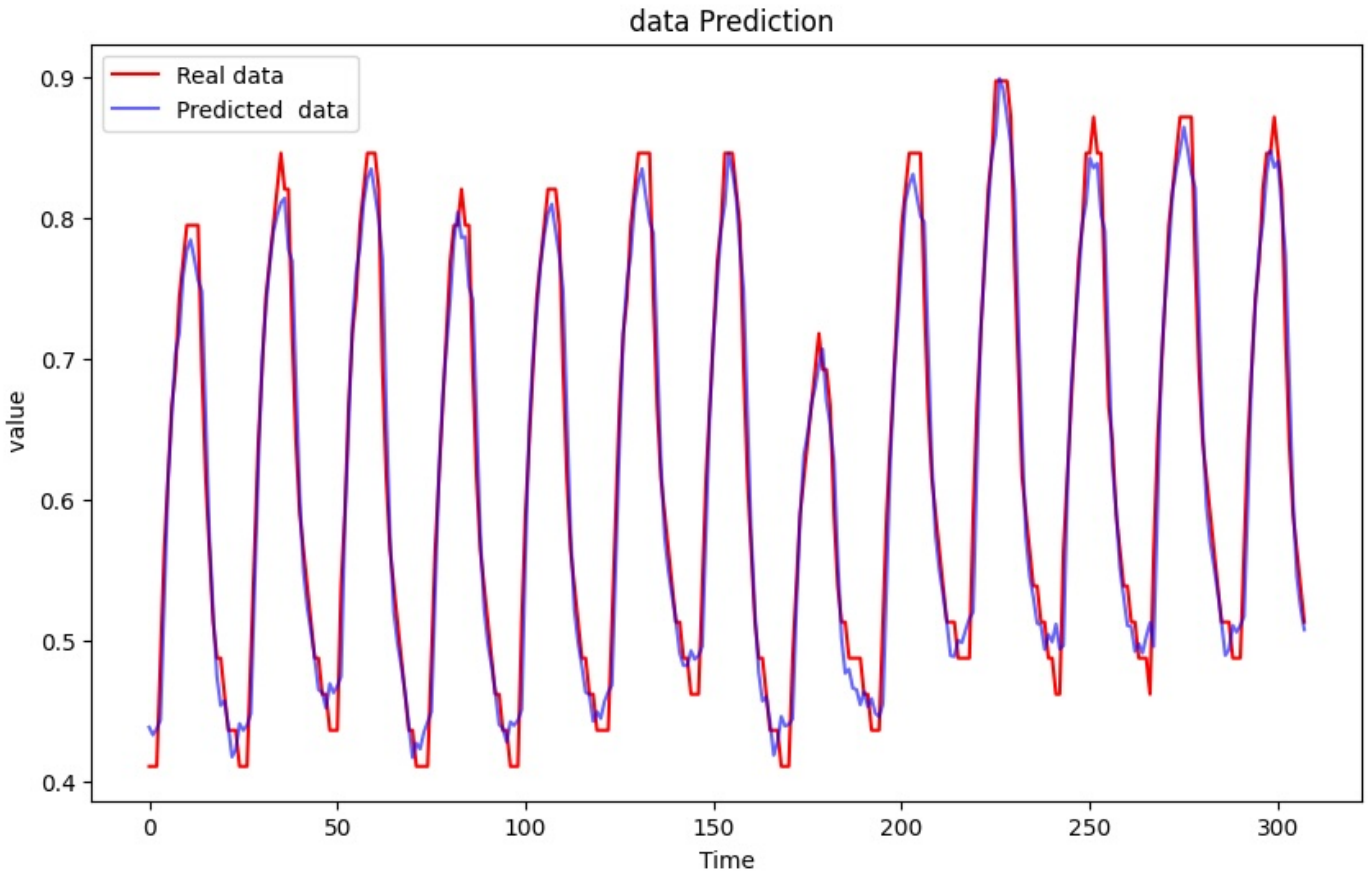
Let's create a predicted array with the test data:

```
prediccion = model.predict(X_test)
```

And plot it:



The trend seems completely captured, with some errors on the peaks. Let's zoom in on the last portion of data only:



The zoomed-in graph provides a more detailed view, where we can observe:

- **Pattern Recognition**: The LSTM model appears to be capturing the cyclical pattern of the data quite well. The peaks and troughs of the predicted values align closely with the actual data, suggesting the model has learned the underlying periodicity in the temperature data.
- **Accuracy**: The close tracking of the predicted values to the actual values, especially the precise capture of the peaks, points to a high degree of accuracy for this segment of the predictions.
- **Phase Shift**: There might be a slight phase shift where the predicted values lead or lag the actual values slightly, a common occurrence in time series forecasting that may require further tuning to correct.
- **Amplitude**: In some parts of the graph, the amplitude of the predicted data (the height of the peaks) does not match perfectly with the real data, which might indicate room for improvement in the model's ability to capture the exact magnitude of changes.

For now, we keep this model, remembering that there is room to improve it in the future.

Create TFLite LSTM Model - Float32

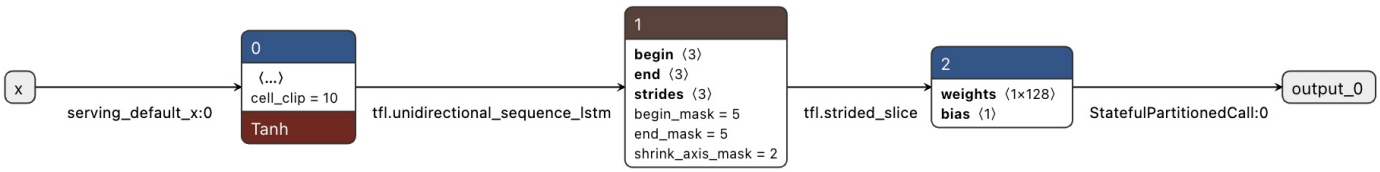
Converting a TensorFlow model to TensorFlow Lite (TFLite) for deployment on microcontrollers (TensorFlow Lite Micro) has some limitations and considerations to keep in mind:

1. **Operator Support**: Not all TensorFlow operations are supported in TFLite, and the support is even more limited for TensorFlow Lite Micro. As of the last update, only `UnidirectionalLSTM` is supported for LSTM operations in TensorFlow Lite Micro, so we need to ensure that our model uses only this type of LSTM layer.
2. **Quantization**: Quantization is the process of reducing the precision of the numbers used to represent a model's parameters, which is essential for running models on devices with limited precision and memory. Although float32 models are supported and tested, quantized models can sometimes present challenges, particularly with TensorFlow Lite Micro, which may not fully support quantization or may not have mature support for all operations in a quantized format. So, we will not use quantization in this project.

```
run_model = tf.function(lambda x: model(x))

BATCH_SIZE = 1
STEPS = n_steps
INPUT_SIZE = n_inputs
concrete_func = run_model.get_concrete_function(
    tf.TensorSpec([BATCH_SIZE, STEPS, INPUT_SIZE], model.inputs[0].dtype))

# model directory.
MODEL_DIR = "keras_lstm"
model.save(MODEL_DIR, save_format="tf", signatures=concrete_func)

converter = tf.lite.TFLiteConverter.from_saved_model(MODEL_DIR)
tflite_model = converter.convert()

# Save the converted model to file
tflite_model_file = 'converted_model.tflite'
with open(tflite_model_file, 'wb') as f:
```

```
    f.write(tflite_model)
```

Here is the converted model, confirming that we only have unidirectional operators:



Deploying the Model with Edge Impulse Python SDK

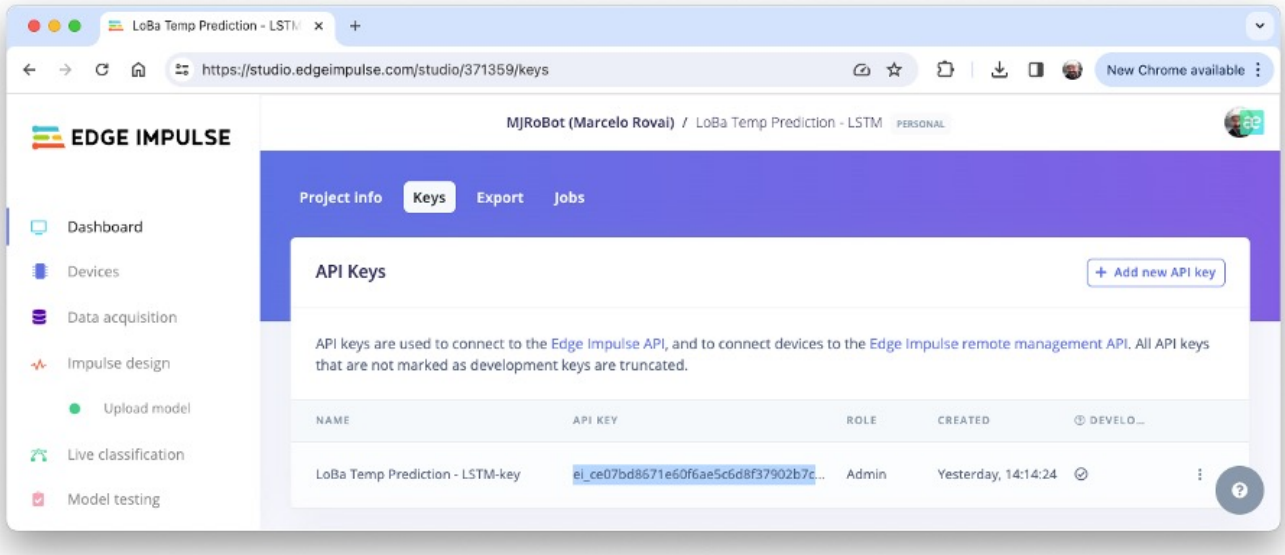
We will need an account with Edge Impulse Studio to deploy our model. If you do not have one, enter the link and follow the instructions.

For this project, we will use the Edge Impulse Python SDK, a library that helps us develop machine learning (ML) applications for edge and Internet of Things (IoT) devices. While the Edge Impulse Studio is a great interface for guiding you through the process of collecting data and training a model, the edgeimpulse Python SDK allows you to programmatically Bring Your Own Model (BYOM), developed and trained on any platform.

First, let us install the Python SDK library:

```
!python -m pip install edgeimpulse
```

To use the Python SDK, you must first create a project in Edge Impulse and copy the API key. Once you have created the project, open it, navigate to **Dashboard**, and click the **Keys** tab to view your API keys. Double-click the API key to highlight it, right-click, and select **Copy**.



> Note that we do not actually will the project in the Edge Impulse Studio. We just need the **API Key**.

From there, import the package and set the API key:

```
import edgeimpulse as ei
ei.API_KEY = "ei_dae27..." # Change to your key
```

Now, we can estimate the RAM, ROM, and inference time for our model (the `tflite_model` that is the `converted_model.tflite`) on the target hardware family, in the case `espressif-esp32`. Once we use an ESP32S3, the latency during inference should be lower than we got here.

```
try:
    profile = ei.model.profile(model=tflite_model,
                               device='espressif-esp32')
    print(profile.summary())
except Exception as e:
    print(f"Could not profile: {e}")
```

Running it, we got the result:

```
Target results for float32:
===========================
{
    "device": "espressif-esp32",
    "tfliteFileSizeBytes": 283024,
    "isSupportedOnMcu": true,
    "memory": {
        "tflite": {
            "ram": 116516,
            "rom": 324136,
            "arenaSize": 116300
        },
        "eon": {
            "ram": 96952,
            "rom": 304936
        }
    },
    "timePerInferenceMs": 31258
}
```

The memory cost for TFLite micro use is estimated in 116 KB of RAM and 324 KB of ROM, what is OK with our device. Note that memory use could be lower using Edge Impulse EON, but we will not use it with LSTM. The issue here seems to be the latency: 32 seconds!!!!! This is very high. Let's see the reality with the ESP32S3.

For deploying our model, we can call the `deploy()` function to convert the model from *tflite* to one of the Edge Impulse-supported outputs. Edge Impulse can output several possible deployment libraries and pre-compiled binaries for various target hardware. In our case, we will use `Arduino`. We should also define the output type, for example, Classification or Regression. In our case, `Regression()`.

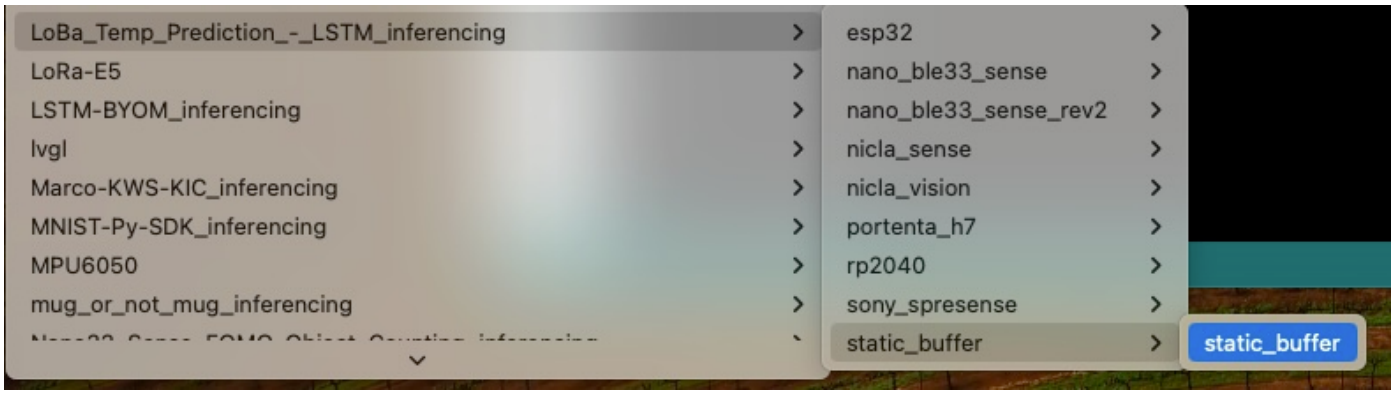
```
download_dir = "./"
deploy_filename = "lstm_float32_model.zip"

# Create an Arduino library with trained model
deploy_bytes = None
try:
    deploy_bytes = ei.model.deploy(tflite_model,
                                   model_output_type=ei.model.output_type.Regression(),
                                   deploy_target='arduino')
except Exception as e:
    print(f"Could not deploy: {e}")

# Write the downloaded raw bytes to a file
if deploy_bytes:
    with open(deploy_filename, 'wb') as f:
        f.write(deploy_bytes.getvalue())
```

Having the created library (`lstm_float32_model.zip`), we should go to the Arduino IDE.

Testing Inference

Open your Arduino IDE, and under `Sketch`, go to `Include Library` and `add.ZIP Library`. Please select the file you create, and that's it!

Under the `Examples` tab on Arduino IDE, you should find a sketch code (static_buffer > static_buffer) under your Edge Impulse project name (in my case: "LoBa_Temp_Prediction-LSTM_Inferencing".



For testing our model using the static buffer sketch, we will need a test datapoint to be loaded on the variable features `static const float features[] = { }`. Note that the input tensor should be "flat" when Edge Impulse deploys the model. So, a datapoint with a shape as **(168, 7)** should be reshaped to **(1176,)**

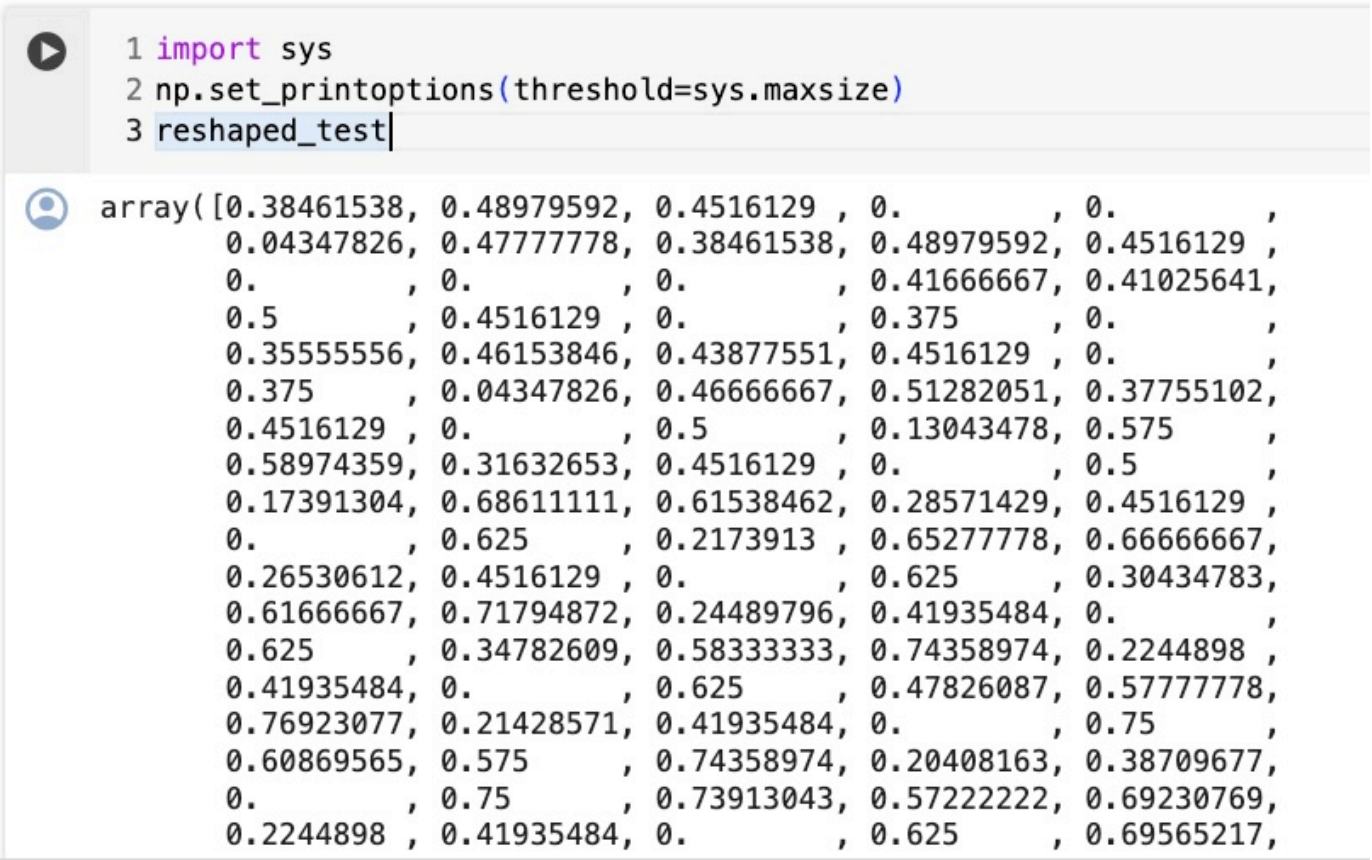
So, let's return to our Notebook and generate a test datapoint:

```
reshaped_test = X_test[0].reshape(-1)
```

Now, let's see all the data:

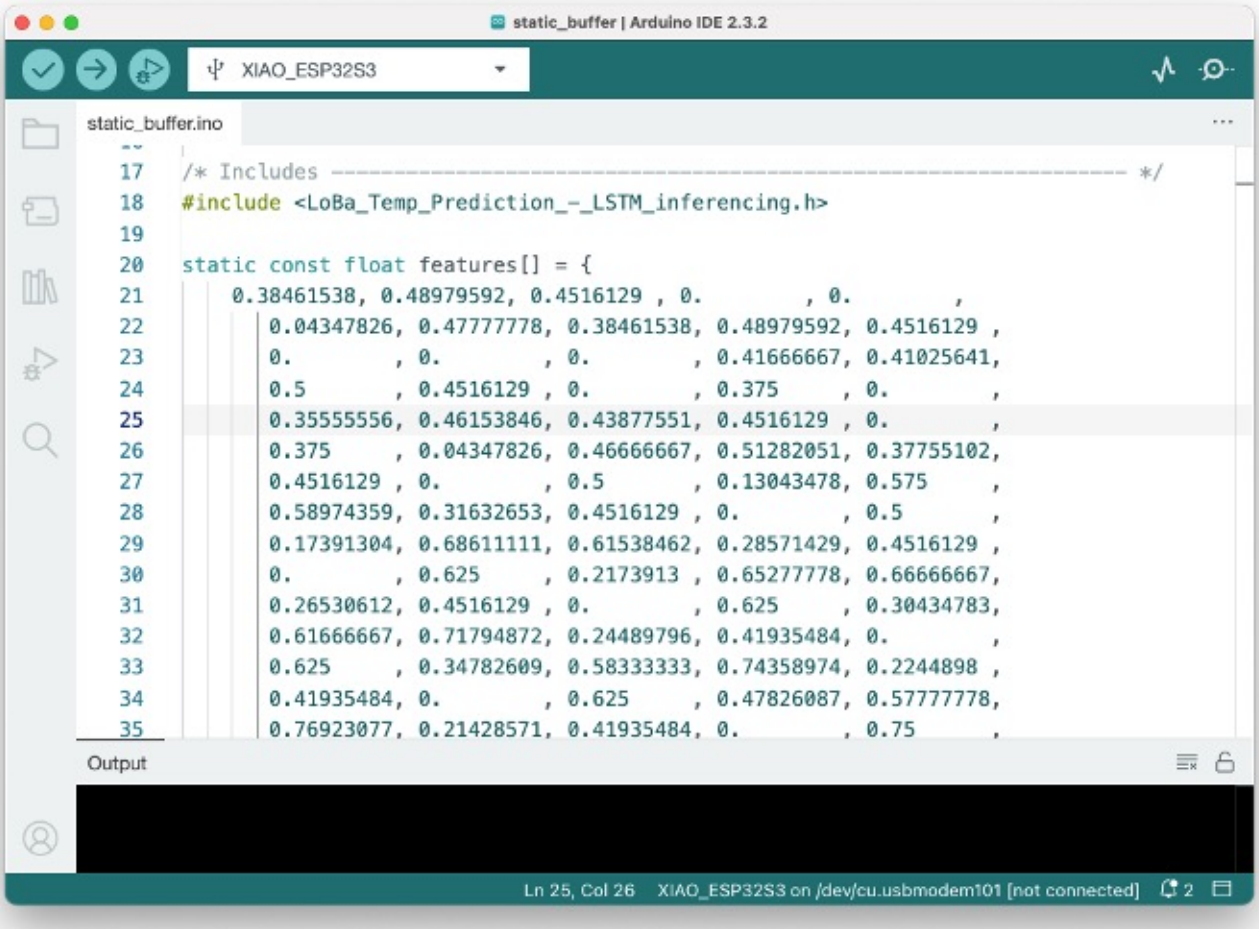
```
import sys
np.set_printoptions(threshold=sys.maxsize)
reshaped_test
```

Copy the values and past them on the Arduino Sketch.



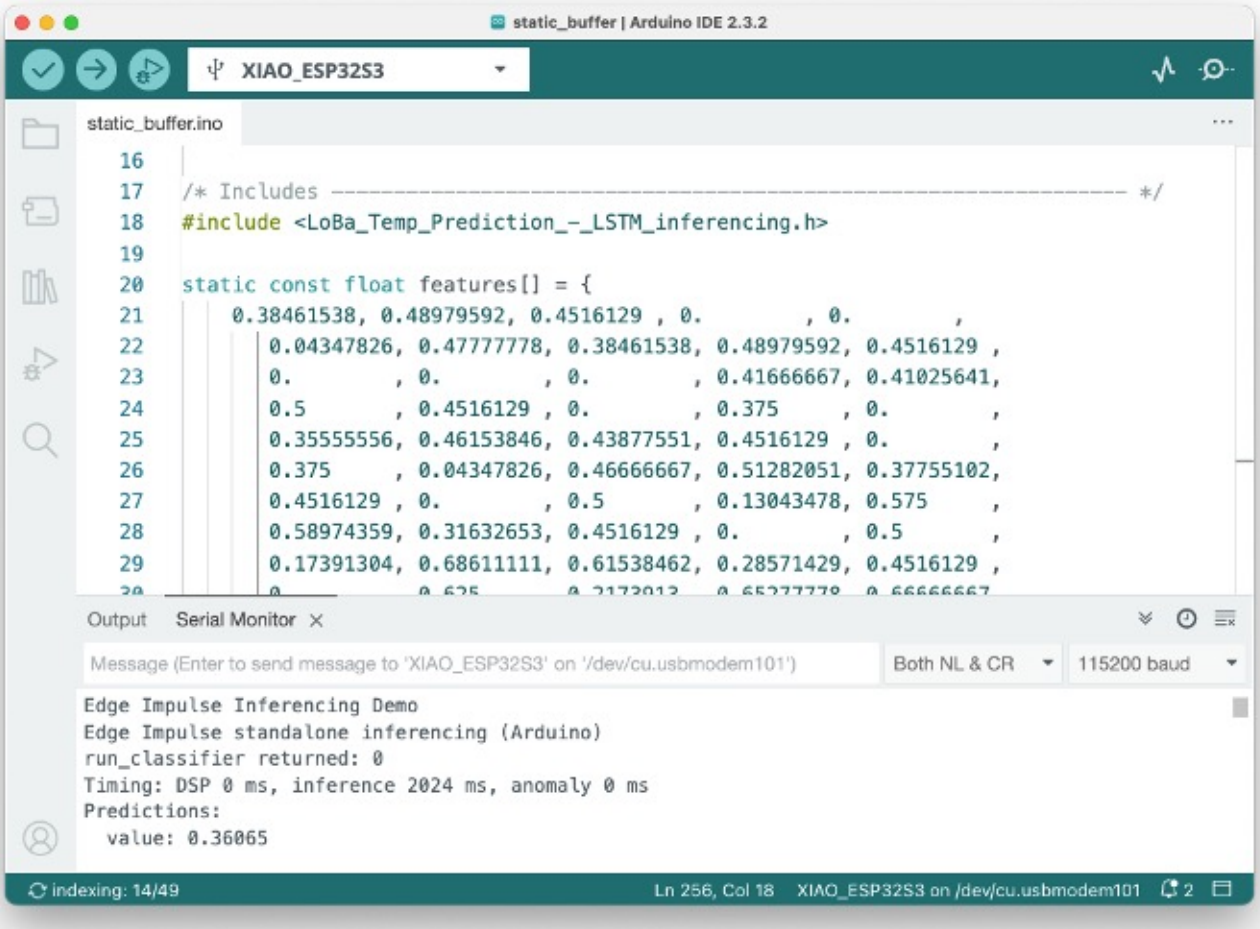
```
1 import sys
2 np.set_printoptions(threshold=sys.maxsize)
3 reshaped_test
```

```
array([0.38461538, 0.48979592, 0.4516129 , 0.        , 0.        ,
       0.04347826, 0.47777778, 0.38461538, 0.48979592, 0.4516129 ,
       0.        , 0.        , 0.        , 0.41666667, 0.41025641,
       0.5       , 0.4516129 , 0.        , 0.375     , 0.        ,
       0.35555556, 0.46153846, 0.43877551, 0.4516129 , 0.        ,
       0.375     , 0.04347826, 0.46666667, 0.51282051, 0.37755102,
       0.4516129 , 0.        , 0.5       , 0.13043478, 0.575     ,
       0.58974359, 0.31632653, 0.4516129 , 0.        , 0.5       ,
       0.17391304, 0.68611111, 0.61538462, 0.28571429, 0.4516129 ,
       0.        , 0.625     , 0.2173913 , 0.65277778, 0.66666667,
       0.26530612, 0.4516129 , 0.        , 0.625     , 0.30434783,
       0.61666667, 0.71794872, 0.24489796, 0.41935484, 0.        ,
       0.625     , 0.34782609, 0.58333333, 0.74358974, 0.2244898 ,
       0.41935484, 0.        , 0.625     , 0.47826087, 0.57777778,
       0.76923077, 0.21428571, 0.41935484, 0.        , 0.75      ,
       0.60869565, 0.575     , 0.74358974, 0.20408163, 0.38709677,
       0.        , 0.75      , 0.73913043, 0.57222222, 0.69230769,
       0.2244898 , 0.41935484, 0.        , 0.625     , 0.69565217,
```

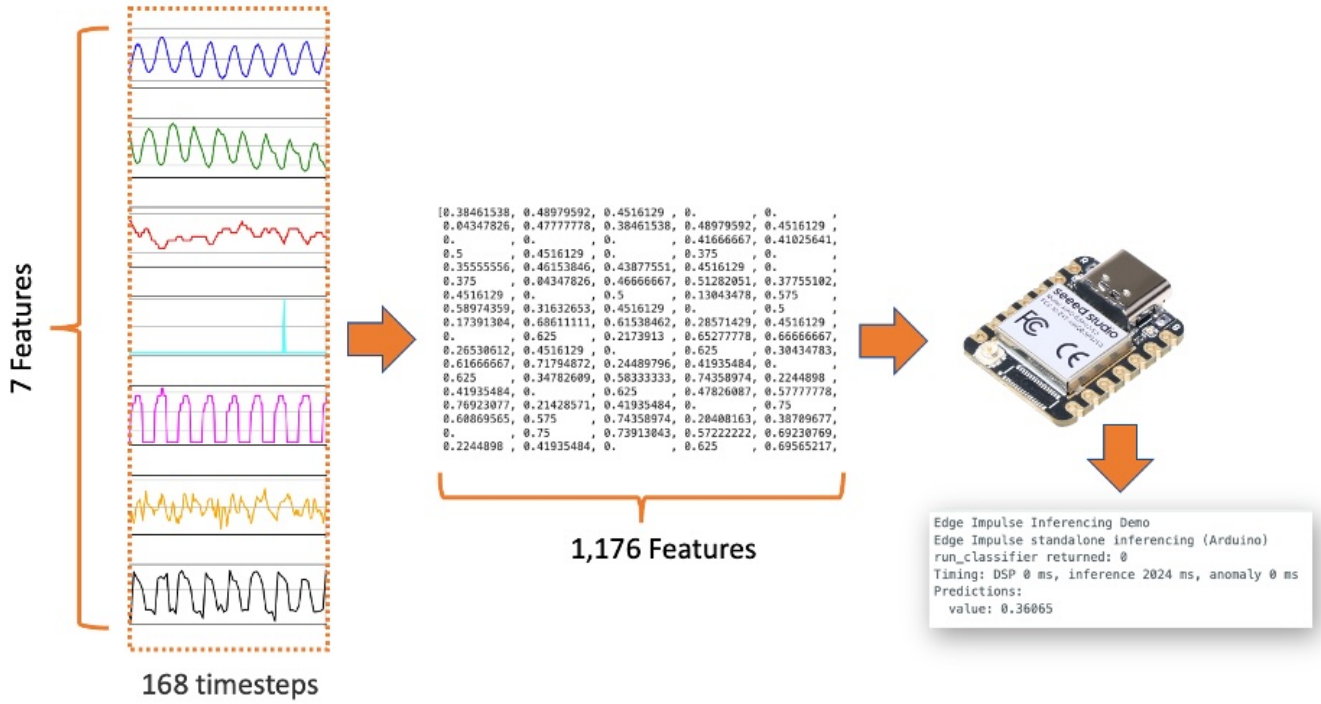


If we inspect the real value of `y_test[0]` we will get 0.3589743589743589

Let's connect our XIAO ESP32S3 and run the schetch. We can see the result on the Serial Monitor:



We can verify that Predict value for this datapoint is 0.36065, what has an error of 0.0017 from the real value! Aslo the latency was around 2 seconds, what is accetable for this project (we will generate one prediction each hour).



We have already verified that an LSTM model works on an embedded device!

Rescaling inference results in real temperature

In a real project, it is important to rescale the inference result to get the value in Centigrade Degrees. After normalization, the parameters are stored in a text file, which can be used to reverse the normalization process and convert our model's predictions back to Celsius.

As a recap, the *Min-Max Scaling* formula is:

$$X_{\text{norm}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \tag{1}$$

And to reverse it:

$$X = X_{\text{norm}} \times (X_{\text{max}} - X_{\text{min}}) + X_{\text{min}} \tag{2}$$

To load these parameters back into a MinMaxScaler instance:

```
loaded_scaler_params = {}
with open('scaler_params.txt', 'r') as file:
    for line in file:
        key, value = line.strip().split(':')
        loaded_scaler_params[key] = np.array([float(i) for i in value.split(',')])
```

And now, we can create, for example, a new scalar instance and set its parameters:

```
inference_scaler = MinMaxScaler()
inference_scaler.scale_ = loaded_scaler_params['scale']
inference_scaler.min_ = loaded_scaler_params['min']
inference_scaler.data_min_ = loaded_scaler_params['data_min']
inference_scaler.data_max_ = loaded_scaler_params['data_max']
inference_scaler.data_range_ = loaded_scaler_params['data_range']
```

The shape of its parameters is (7) because `MinMaxScaler()` was applied to all 7 input features. Let's take only the min and max values for the temperature (the first feature):

```
data_min = inference_scaler.data_min_[0]
data_max = inference_scaler.data_max_[0]
```

> We can also use `data_min` and `data_range`, once `data_range = data_max - data_min`.

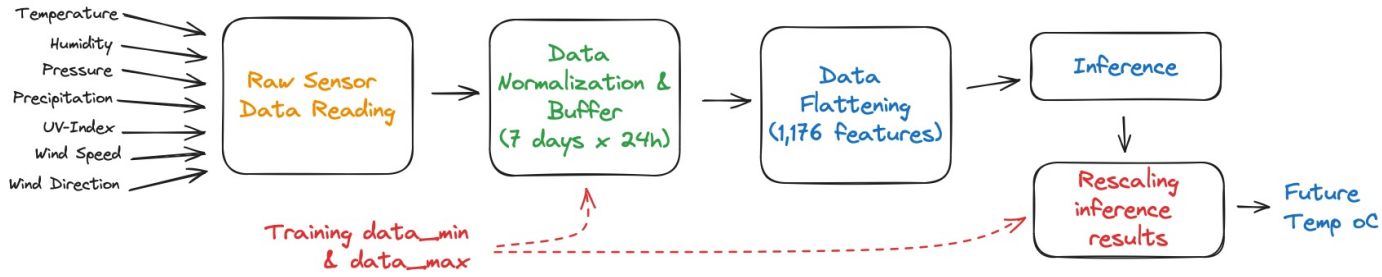
Reverse Normalization for Predictions

Applying the function (2) above to the inference result `value` of **0.36065**, we get as `temp`, **10 degrees Celcius.**

```
# Convert normalized predictions back to Celsius
value = 0.36065
temp = value * (data_max - data_min) + data_min
```

Further work

From this point, to develop a real project for predicting temperature, we should get data from the sensors (sensor integration and reading), data buffering, normalization, and flattening.

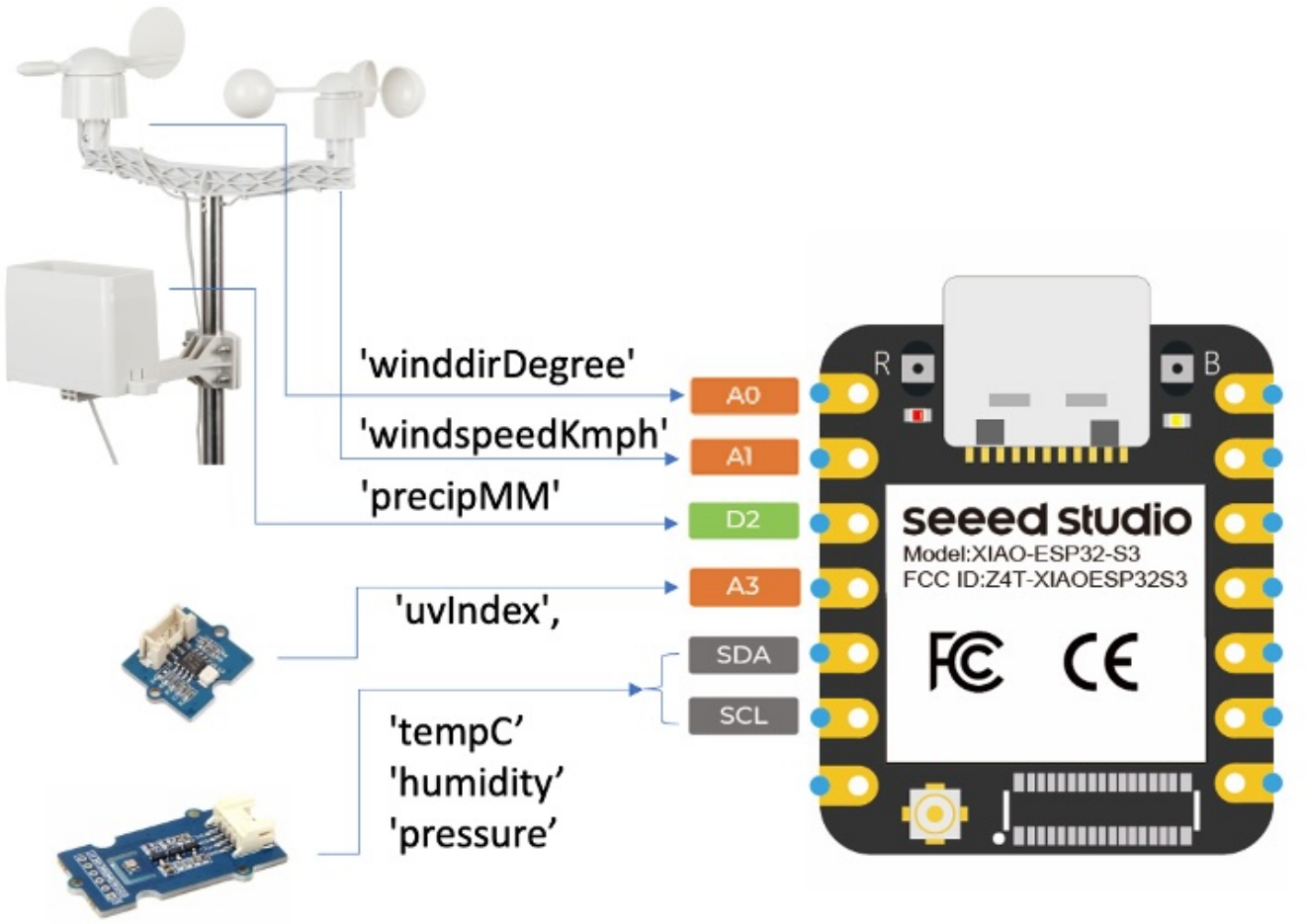


> We will not implement the project but highlight its most important parts, giving directions for its development.

Sensor Integration

We can use the ESP32S3's ADC channels, digital I/Os, and I2C interface to collect sensor data. For example:

- Analog input pins to get wind direction (A0) and speed (A1) from sensors
- Digital pin for Bucket rain Gauge (D2)
- The UV sensor should use an analog (A3).
- Use I2C for Pressure, Temperature, and Humidity - BME280 (SDA/SCL).



Sensor Data Reading

The raw sensor data should be read and converted to the appropriate scales:

- Wind Direction: From voltage to Integer degrees
- Wind Speed: From voltage to Integer Kilometer per hour
- Precipitation: From digital pulse count to float millimeter
- UV Index: From voltage to Integer index
- Temperature, Humidity, and Pressure can be read directly on its scales.

Let's do it in parts:

First, we need to read the raw data from the sensors and convert them to the scales used during the model training. Here is how we can approach it:

Wind Direction and Speed (Analog Input):

```
// Wind direction is a value from 0 to 360 degrees
int readWindDirection() {
  int windDirectionRaw = analogRead(A0);
  // Conversion from raw reading to degrees should be calibrated
  // based on your specific sensor and its voltage-to-angle mapping
  int windDirectionDegrees = map(windDirectionRaw, 0, 4095, 0, 360);
  return windDirectionDegrees;
```

```
}

// Wind speed is a value in kilometers per hour
int readWindSpeed() {
  int windSpeedRaw = analogRead(A1);
  // Conversion from raw reading to speed should be calibrated
  // based on your specific sensor and its voltage-to-speed mapping
  int windSpeedKmph = map(windSpeedRaw, 0, 4095, 0, max_wind_speed);
  return windSpeedKmph;
}
```

Rain Gauge (Digital Input):

```
// Setup the rain gauge interrupt
void setup() {
  pinMode(D2, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(D2), rainGaugeISR, FALLING);
}

// ISR for the rain gauge to count bucket tips
void rainGaugeISR() {
  bucketTips++;
}

// Calculate precipitation in millimeters
float calculatePrecipitation() {
  float precipMM = bucketTips * bucket_tip_amount; // bucket_tip_amount should be the
amount of rain per bucket tip
  bucketTips = 0; // Reset the counter after the reading
  return precipMM;
}
```

UV Index (Analog Input):

```
int readUVIndex() {
  int uvRaw = analogRead(A3);
  // Convert the raw UV reading to an index
  // Calibration required based on sensor specifics
  int uvIndex = map(uvRaw, 0, 4095, 0, max_uv_index);
  return uvIndex;
}
```

> Note: in the tutorial IoT Made Easy: Capturing Remote Weather Data, you can see a more detailed code for a UV sensor read

BME280 (I2C Temperature, Humidity, and Pressure):

```
#include <Wire.h>
#include "Seeed_BME280.h"

void setup() {
  Wire.begin();
  bme.begin(0x76); // I2C address for BME280
}

void readBME280(int &temperature, int &humidity, int &pressure) {
  temperature = bme.getTemperature();
  humidity = bme.getHumidity();
  pressure = bme.getPressure() / 100.0F; // Convert to hPa
}
```

Buffer Sensor Data

When we read the data from each sensor, we must store it in a buffer that keeps track of the last seven days' worth of data. Considering a one-hour sampling rate and needing seven days' worth of data, we would have 168 samples per sensor.

```
// Constants
const int numSensors = 7;
const int samplesPerDay = 24;
const int daysToBuffer = 7;
const int totalSamples = samplesPerDay * daysToBuffer;

// Buffers for each sensor data
float sensorBuffers[numSensors][totalSamples];

// Current position in each buffer
int bufferIndices[numSensors] = {0};

/*
   The newSensorData[] should keep the same data sequency used during training:
   ['tempC', 'humidity', 'pressure', 'precipMM', 'uvIndex', 'windspeedKmph',
   'winddirDegree']
*/

// Function to read data from each sensor and update the buffer
void updateBuffers() {
  // Read new data from each sensor
  float newSensorData[numSensors];
  newSensorData[0] = temperature; // Replace with actual function calls
  newSensorData[1] = humidity;     // and so on for each sensor...
  // ... populate newSensorData for other sensors

  // Update the buffers with the new data
  for (int i = 0; i < numSensors; i++) {
    sensorBuffers[i][bufferIndices[i]] = newSensorData[i];
    bufferIndices[i] = (bufferIndices[i] + 1) % totalSamples;
```

```
  }
}
```

We should call `updateBuffers()` once every hour. We can use the `millis()` function to check if an hour has passed since the last collection, then read and buffer new data.

```
unsigned long lastReadTime = 0;
const unsigned long readInterval = 3600000; // One hour in milliseconds

void loop() {
  if (millis() - lastReadTime >= readInterval
```

Normalize the Data

Normalization is done based on each sensor type's pre-recorded minimum and maximum values. You'll normalize the data as you read it before buffering.

```
float minValues[numSensors] = { /* ... min values for each sensor ... */ };
float maxValues[numSensors] = { /* ... max values for each sensor ... */ };

// Normalize a value based on its sensor's min/max
float normalize(float value, int sensorIndex) {
  return (value - minValues[sensorIndex]) / (maxValues[sensorIndex] -
minValues[sensorIndex]);
}

// Update normalization as you buffer
// Inside your updateBuffers() function, before assigning newSensorData to the buffer:
for (int i = 0; i < numSensors; i++) {
  sensorBuffers[i][bufferIndices[i]] = normalize(newSensorData[i], i);
}
```

Flatten the Data for Inference

At this point, our sensorBuffers[] are loaded with the 7 days of data (from each of 7 sensors). Before performing inference, we should flatten the buffered data into a single array that can be inputted into the LSTM model:

```
static const float features[totalSamples * numSensors]; // 168 * 7 = 1,176 features

void flattenDataForModel() {
  for (int i = 0; i < numSensors; i++) {
    for (int j = 0; j < totalSamples; j++) {
      int flatIndex = i * totalSamples + j;
      features[flatIndex] = sensorBuffers[i][(bufferIndices[i] + j) % totalSamples];
    }
  }
}
```

We should keep a rolling buffer for each sensor's data so the buffer will always have the last seven days of data. Having the features populated, we can calculate the inference based on the Edge Impulse Static buffer code example, shown below:

```
/* Includes ---------------------------------------------------------------- */
#include <LoBa_Temp_Prediction_-_LSTM_inferencing.h>

static const float features[] = {
    // copy raw features here (for example from the 'Live classification' page)
    // see https://docs.edgeimpulse.com/docs/running-your-impulse-arduino
};

/**
 * @brief      Copy raw feature data in out_ptr
 *             Function called by inference library
 *
 * @param[in]  offset   The offset
 * @param[in]  length   The length
 * @param      out_ptr  The out pointer
 *
 * @return     0
 */
int raw_feature_get_data(size_t offset, size_t length, float *out_ptr) {
    memcpy(out_ptr, features + offset, length * sizeof(float));
    return 0;
}

void print_inference_result(ei_impulse_result_t result);

/**
 * @brief      Arduino setup function
 */
void setup()
{
    // put your setup code here, to run once:
    Serial.begin(115200);
    // comment out the below line to cancel the wait for USB connection (needed for native
USB)
    while (!Serial);
    Serial.println("Edge Impulse Inferencing Demo");
}

/**
 * @brief      Arduino main function
 */
void loop()
{
    ei_printf("Edge Impulse standalone inferencing (Arduino)\n");

    if (sizeof(features) / sizeof(float) != EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE) {
```

```
        ei_printf("The size of your 'features' array is not correct. Expected %lu items,
but had %lu\n",
            EI_CLASSIFIER_DSP_INPUT_FRAME_SIZE, sizeof(features) / sizeof(float));
        delay(1000);
        return;
    }

    ei_impulse_result_t result = { 0 };

    // the features are stored into flash, and we don't want to load everything into RAM
    signal_t features_signal;
    features_signal.total_length = sizeof(features) / sizeof(features[0]);
    features_signal.get_data = &raw_feature_get_data;

    // invoke the impulse
    EI_IMPULSE_ERROR res = run_classifier(&features_signal, &result, false /* debug */);
    if (res != EI_IMPULSE_OK) {
        ei_printf("ERR: Failed to run classifier (%d)\n", res);
        return;
    }

    // print inference return code
    ei_printf("run_classifier returned: %d\r\n", res);
    print_inference_result(result);

    delay(1000);
}

void print_inference_result(ei_impulse_result_t result) {

    // Print how long it took to perform inference
    ei_printf("Timing: DSP %d ms, inference %d ms, anomaly %d ms\r\n",
            result.timing.dsp,
            result.timing.classification,
            result.timing.anomaly);

    // Print the prediction results (object detection)
#if EI_CLASSIFIER_OBJECT_DETECTION == 1
    ei_printf("Object detection bounding boxes:\r\n");
    for (uint32_t i = 0; i < result.bounding_boxes_count; i++) {
        ei_impulse_result_bounding_box_t bb = result.bounding_boxes[i];
        if (bb.value == 0) {
            continue;
        }
        ei_printf("  %s (%f) [ x: %u, y: %u, width: %u, height: %u ]\r\n",
                bb.label,
                bb.value,
                bb.x,
                bb.y,
                bb.width,
                bb.height);
    }
```

```
    // Print the prediction results (classification)
#else
    ei_printf("Predictions:\r\n");
    for (uint16_t i = 0; i < EI_CLASSIFIER_LABEL_COUNT; i++) {
        ei_printf("  %s: ", ei_classifier_inferencing_categories[i]);
        ei_printf("%.5f\r\n", result.classification[i].value);
    }
#endif

    // Print anomaly result (if it exists)
#if EI_CLASSIFIER_HAS_ANOMALY == 1
    ei_printf("Anomaly prediction: %.3f\r\n", result.anomaly);
#endif

}
```

The inference results `result.classification[0].value` should be rescaled to convert normalized predictions back to Celsius:

```
value = result.classification[0].value
temp = value * (maxValues[0] - (minValues[0]) + minValues[0]
```

Conclusion

In this tutorial, we've explored how to implement a temperature prediction system for Lo Barnechea, Chile, using an LSTM neural network model. We covered the full cycle of a temperature prediction project using LSTM, from data acquisition to deploying the model for real-time predictions. The LSTM model's strength lies in its ability to recognize patterns in time-series data, making it suitable for predicting future temperature values based on historical sensor data.

We learned the importance of the data preparation phase, where the raw data was cleaned, normalized, and structured so that the LSTM model could understand and learn from it. This step was crucial once the quality and format of the input data significantly impacted the model's ability to learn and make accurate predictions.

Once the model had learned from the historical data, it was converted using the Edge Impulse Python SDK library into a format suitable for the deployment environment—in this case, TensorFlow Lite for microcontrollers.

The final step was deployment, where the converted model was loaded onto the XIAO ESP32-S3 microcontroller. The model's deployment marked a significant milestone—transforming theory into practice. The XIAO ESP32-S3, equipped with the LSTM model, was empowered to analyze real-time sensor data spanning temperature, humidity, pressure, and other climatic variables to predict impending temperature changes. We leveraged the ESP32-S3's capabilities, such as ADC channels, digital I/Os, and the I2C protocol, to establish a seamless and reliable data collection routine.

By interweaving sensor technology, data science, and software engineering, this project stands as a testament to the potential of machine learning to enhance our understanding and anticipation of natural phenomena. The practical applications of such a temperature prediction system are broad and impactful, ranging from environmental monitoring and smart agriculture to urban planning—each benefitting from the foresight that accurate temperature predictions can provide.

This tutorial underscores the end-to-end methodology required to bring machine learning models from concept to reality in an embedded system context. It celebrates the advancements in embedded AI that enable devices like the XIAO ESP32-S3 to perform complex tasks like time series forecasting with LSTM models—once the domain of high-powered computing systems—demonstrating the increasing convergence of sophisticated AI and everyday devices.