Climate Data Time-Series

You are again moving to another role, not at *The Weather Channel*, where you are ask to create a Weather Forecasting Model.

For that, you will be using *Jena Climate* dataset recorded by the *Max Planck Institute for Biogeochemistry*.

The dataset consists of 14 features such as temperature, pressure, humidity etc, recorded **once per 10 minutes**.

Location: Weather Station, Max Planck Institute for Biogeochemistry in Jena, Germany

Time-frame Considered: **Jan 10, 2009 - December 31, 2012**

Library Imports

In [20]:
```
import pandas as pd
import matplotlib.pyplot as plt
import keras
```

1) Load your data

Your data can be found on the Deep Learning Module under a file named: climate_data_2009_2012.csv

In [21]:
```
df = pd.read_csv("climate_data_2009_2012.csv")
```

2) Data engineering

You are given 3 lists:

- titles: Display names of your columns
- feature_keys: Names of the columns used as features
- colors: The color to use when ploting that column's value

In [22]:
```
titles = [
    "Pressure",
    "Temperature",
    "Temperature in Kelvin",
```

```
    "Temperature (dew point)",
    "Relative Humidity",
    "Saturation vapor pressure",
    "Vapor pressure",
    "Vapor pressure deficit",
    "Specific humidity",
    "Water vapor concentration",
    "Airtight",
    "Wind speed",
    "Maximum wind speed",
    "Wind direction in degrees",
]

feature_keys = [
    "p (mbar)",
    "T (degC)",
    "Tpot (K)",
    "Tdew (degC)",
    "rh (%)",
    "VPmax (mbar)",
    "VPact (mbar)",
    "VPdef (mbar)",
    "sh (g/kg)",
    "H2OC (mmol/mol)",
    "rho (g/m**3)",
    "wv (m/s)",
    "max. wv (m/s)",
    "wd (deg)",
]

colors = [
    "blue",
    "orange",
    "green",
    "red",
    "purple",
    "brown",
    "pink",
    "gray",
    "olive",
    "cyan",
]
```

Let's look at the climate data:

In [23]:
```
df.head()
```

Out[23]:

	Date Time	p (mbar)	T (degC)	Tpot (K)	Tdew (degC)	rh (%)	VPmax (mbar)	VPact (mbar)	VPdef (mbar)	sh (g/kg)
0	01.01.2009 00:10:00	996.52	-8.02	265.40	-8.90	93.3	3.33	3.11	0.22	1.94
1	01.01.2009 00:20:00	996.57	-8.41	265.01	-9.28	93.4	3.23	3.02	0.21	1.89
2	01.01.2009 00:30:00	996.53	-8.51	264.91	-9.31	93.9	3.21	3.01	0.20	1.88
3	01.01.2009 00:40:00	996.51	-8.31	265.12	-9.07	94.2	3.26	3.07	0.19	1.92
4	01.01.2009 00:50:00	996.51	-8.27	265.15	-9.04	94.1	3.27	3.08	0.19	1.92

Define a function to show a plot of each column (using the respective color)

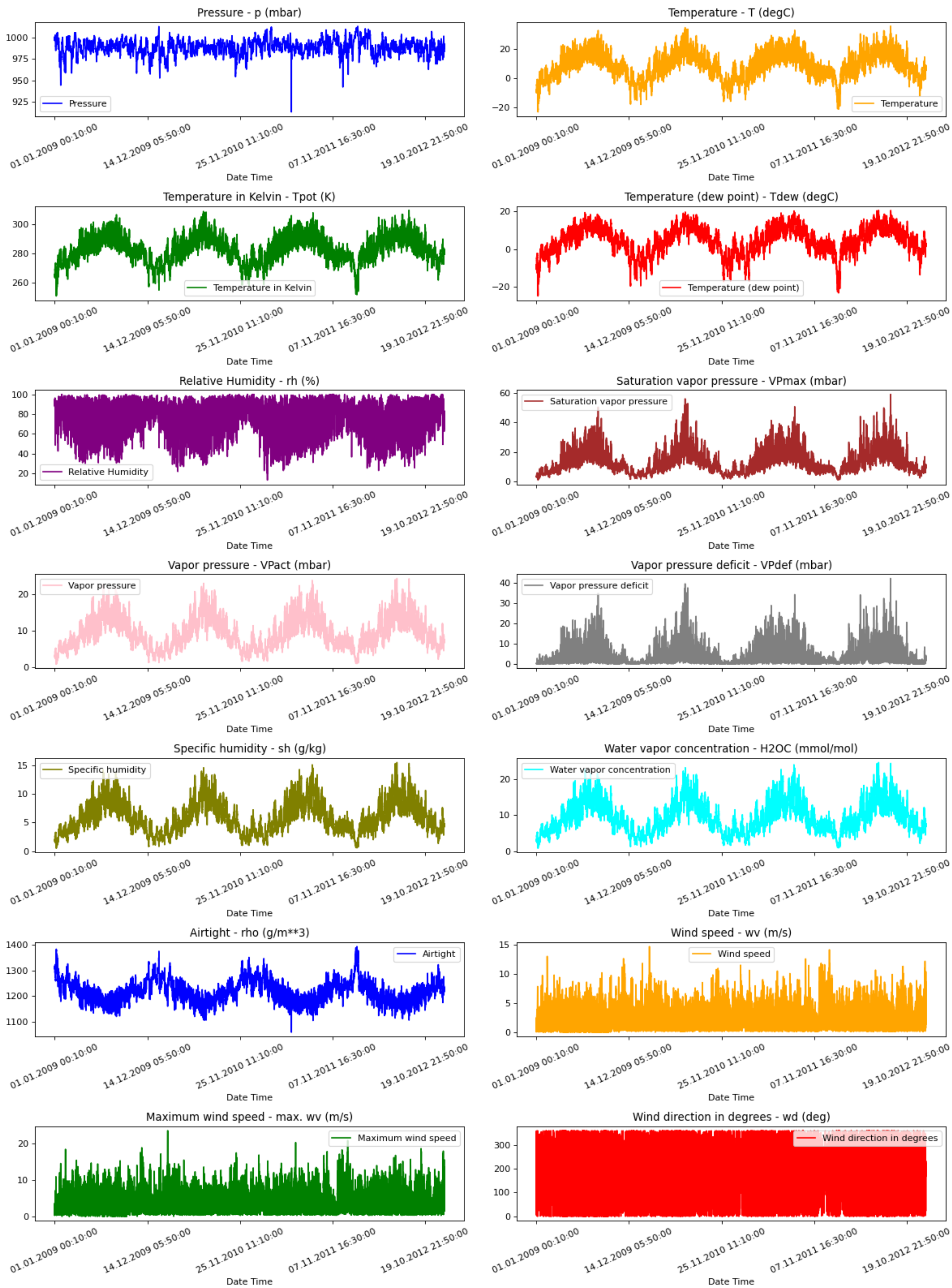
In [24]:

```
def show_raw_visualization(data, date_time_key):
    time_data = data[date_time_key]
    fig, axes = plt.subplots(
        nrows=7, ncols=2, figsize=(15, 20), dpi=80, facecolor="w", edgecolor
    )
    for i in range(len(feature_keys)):
        key = feature_keys[i]
        c = colors[i % (len(colors))]
        t_data = data[key]
        t_data.index = time_data
        t_data.head()
        ax = t_data.plot(
            ax=axes[i // 2, i % 2],
            color=c,
            title="{} - {}".format(titles[i], key),
            rot=25,
        )
        ax.legend([titles[i]])
    plt.tight_layout()
```

Display each column in a plot using above funciton:

In [25]:

```
show_raw_visualization(df, "Date Time")
```



As you can see we have lots of data, this can be a challenge when we train our model, to resolve that we will reduce the resolution of our data, instead of having a climate signal each 10 minutes, we will have it each hour

- Add a new column to your dataframe with the Date Time information
- Name that column FormatedDateTime
- Convert that column into date time data type
- Set that column as the dataframe index
- Regroup data to be each 1 hour instead of each 10 minutes
- Save the grouped data into a dataframe called df_resampled
- Remove the FormatedDateTime as the index.
- Show the top 5 rows of df_resampled

In [26]:
```
df['FormatedDateTime'] = pd.to_datetime(df['Date Time'], format='%d.%m.%Y %H
df = df.set_index('FormatedDateTime')
df_resampled = df[feature_keys].resample('H').mean()
df_resampled = df_resampled.reset_index()

df_resampled.head()
```

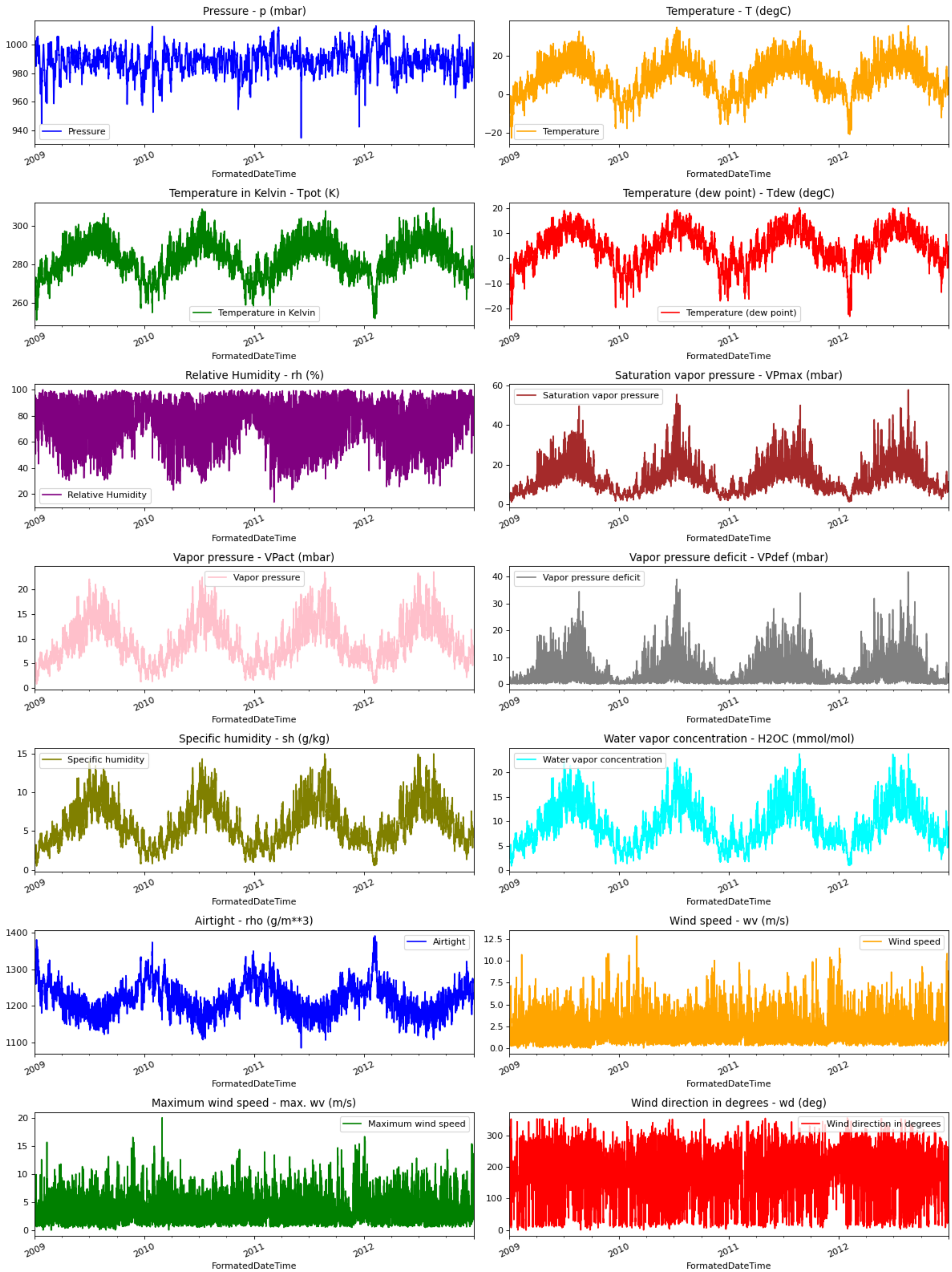
```
/var/folders/cv/1yfc2xyj17z1yx0z1swfxbdw0000gn/T/ipykernel_17687/3134505598.
py:3: FutureWarning: 'H' is deprecated and will be removed in a future versi
on, please use 'h' instead.
  df_resampled = df[feature_keys].resample('H').mean()
```

Out[26]:

	FormatedDateTime	p (mbar)	T (degC)	Tpot (K)	Tdew (degC)	rh (%)	
0	2009-01-01 00:00:00	996.528000	-8.304000	265.118000	-9.120000	93.780000	3.
1	2009-01-01 01:00:00	996.525000	-8.065000	265.361667	-8.861667	93.933333	3.
2	2009-01-01 02:00:00	996.745000	-8.763333	264.645000	-9.610000	93.533333	3.
3	2009-01-01 03:00:00	996.986667	-8.896667	264.491667	-9.786667	93.200000	3
4	2009-01-01 04:00:00	997.158333	-9.348333	264.026667	-10.345000	92.383333	3

Let's look at our fields again

In [27]:
```
show_raw_visualization(df_resampled, "FormatedDateTime")
```


Pressure - p (mbar)
Pressure
Temperature - T (degC)
Temperature
Temperature in Kelvin - Tpot (K)
Temperature in Kelvin
Temperature (dew point) - Tdew (degC)
Temperature (dew point)
Relative Humidity - rh (%)
Relative Humidity
Saturation vapor pressure - VPmax (mbar)
Saturation vapor pressure
Vapor pressure - VPact (mbar)
Vapor pressure
Vapor pressure deficit - VPdef (mbar)
Vapor pressure deficit
Specific humidity - sh (g/kg)
Specific humidity
Water vapor concentration - H2OC (mmol/mol)
Water vapor concentration
Airtight - rho (g/m**3)
Airtight
Wind speed - wv (m/s)
Wind speed
Maximum wind speed - max. wv (m/s)
Maximum wind speed
Wind direction in degrees - wd (deg)
Wind direction in degrees
2009
2010
2011
2012
FormatedDateTime

3) Data Split: Train and Evaluation datasets.

- We are tracking data from past 120 timestamps (120 hours = 5 days).
- This data will be used to predict the temperature after 12 timestamps (12 hours).
- Since every feature has values with varying ranges, we do normalization to confine feature values to a range of [0, 1] before training a neural network.
- We do this by subtracting the mean and dividing by the standard deviation of each feature in the *normalize* function
- The model is shown data for first 5 days i.e. 120 observations, that are sampled every hour.
- The temperature after 12 hours observation will be used as a label.

In [28]:
```
# 70% of the data will be used for training, the rest for testing
split_fraction = 0.7
# The number of samples is the number of rows in the data
number_of_samples = df_resampled.shape[0]
# The size in rows of the split dataset
train_split = int(split_fraction * int(number_of_samples))

# Number of samples in the past used to predict the future
past = 120
# Number of samples in the future to predict (the value in the 72nd hour is
future = 12
# Learning rate parameter for the Adam optimizer
learning_rate = 0.001
# Batch size for the model training
batch_size = 256
# Number of epochs for the model training
epochs = 10

# Another way to normalize the data (all columns in the same range)
def normalize(data, train_split):
    data_mean = data[:train_split].mean(axis=0)
    data_std = data[:train_split].std(axis=0)
    return (data - data_mean) / data_std
```

- Let's select the following parameters as our features:
  - Pressure, Temperature, Saturation vapor pressure, Vapor pressure deficit, Specific humidity, Airtight, Wind speed
- Set the column FormatedDateTime as the index of our dataframe.
  - This is important since now, FormatedDateTime is used as our datetime field and not as a Feature field

- Normalize all fields
- Generate two datasets:
  - train_data: Train dataset with our normalized fields
  - val_data: Validation dataset

In [29]:
```
print(
    "The selected parameters are:",
    ", ".join([titles[i] for i in [0, 1, 5, 7, 8, 10, 11]]),
)
selected_features = [feature_keys[i] for i in [0, 1, 5, 7, 8, 10, 11]]
features = df_resampled[selected_features]
features.index = df_resampled["FormatedDateTime"]
print(features.head())

features = normalize(features.values, train_split)
features = pd.DataFrame(features)
print(features.head())

train_data = features.loc[0 : train_split - 1]
val_data = features.loc[train_split:]
```

```
The selected parameters are: Pressure, Temperature, Saturation vapor pressur
e, Vapor pressure deficit, Specific humidity, Airtight, Wind speed
                              p (mbar)  T (degC)  VPmax (mbar)  VPdef (mbar)  \
FormatedDateTime
2009-01-01 00:00:00  996.528000 -8.304000      3.260000      0.202000
2009-01-01 01:00:00  996.525000 -8.065000      3.323333      0.201667
2009-01-01 02:00:00  996.745000 -8.763333      3.145000      0.201667
2009-01-01 03:00:00  996.986667 -8.896667      3.111667      0.210000
2009-01-01 04:00:00  997.158333 -9.348333      3.001667      0.231667

                     sh (g/kg)  rho (g/m**3)  wv (m/s)
FormatedDateTime
2009-01-01 00:00:00   1.910000   1309.196000  0.520000
2009-01-01 01:00:00   1.951667   1307.981667  0.316667
2009-01-01 02:00:00   1.836667   1311.816667  0.248333
2009-01-01 03:00:00   1.811667   1312.813333  0.176667
2009-01-01 04:00:00   1.733333   1315.355000  0.290000
          0         1         2         3         4         5         6
0  0.988366 -1.936957 -1.314750 -0.797292 -1.472751  2.198783 -1.116409
1  0.988002 -1.909978 -1.306369 -0.797363 -1.457136  2.169559 -1.256715
2  1.014643 -1.988807 -1.329968 -0.797363 -1.500234  2.261854 -1.303867
3  1.043907 -2.003858 -1.334379 -0.795594 -1.509604  2.285840 -1.353320
4  1.064694 -2.054843 -1.348935 -0.790994 -1.538961  2.347009 -1.275116
```

Now, here we need to set our Label Dataset.

- We want to use the last 5 days of data, to predict the next 12 hours
- This means that our label starts at the 12th hour after the history data.
    - [.......................... .]
    - -----Start----->
- And it will end at the end of our train dataset size.
    - <-------- Train ---------> <--- Test --->
    - [..|...........]
    - ----------End---------->

In [30]:
```
start = past + future
end = start + train_split

x_train = train_data[[i for i in range(7)]].values
y_train = features.iloc[start:end][[1]]

step = 1
sequence_length = past
```

The *timeseries_dataset_from_array* function takes in a sequence of data-points gathered at equal intervals, along with time series parameters such as length of the sequences/windows, spacing between two sequence/windows, etc., to produce batches of sub-timeseries inputs and targets sampled from the main timeseries.

- Input data (hour features) = x_train
- The **corresponding** value of the temperature 12 hours into the future = y_train
- Since we want to use 5 days of data to predict the future temperature then: sequence_length = 120
- Since we want to sample every hour then: sampling_rate = 1
- Let's use a common batch size of 256 (variable above)

In [31]:
```
dataset_train = keras.preprocessing.timeseries_dataset_from_array(
    x_train,
    y_train,
    sequence_length=sequence_length,
    sampling_rate=step,
    batch_size=batch_size,
)
```

Now let's prepare our validation dataset:

- The validation dataset must not contain the last 120+12 rows as we won't have label data for those records, hence these rows must be subtracted from the end of the data.

- The validation label dataset must start from 120+12 after train_split, hence we must add past + future to label_start.

In [32]:
```
x_end = len(val_data) - past - future

label_start = train_split + past + future

x_val = val_data.iloc[:x_end][[i for i in range(7)]].values
y_val = features.iloc[label_start:][[1]]

dataset_val = keras.preprocessing.timeseries_dataset_from_array(
    x_val,
    y_val,
    sequence_length=sequence_length,
    sampling_rate=step,
    batch_size=batch_size,
)


for batch in dataset_train.take(1):
    inputs, targets = batch

print("Input shape:", inputs.numpy().shape)
print("Target shape:", targets.numpy().shape)
```

```
Input shape: (256, 120, 7)
Target shape: (256, 1)
```

```
2024-10-17 07:16:38.257920: I tensorflow/core/framework/local_rendezvous.cc:
404] Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
```

4) Define and Compile your model:

- An input layer
- A Long Short-Term Memory Hidden Layer with 32 units. LSTM is a type of recurrent neural network layer that is well-suited for time series data.
- An output Dense Layer (Linear Activation function)

In [33]:
```
inputs = keras.layers.Input(shape=(inputs.shape[1], inputs.shape[2]))
lstm_out = keras.layers.LSTM(32)(inputs)
outputs = keras.layers.Dense(1)(lstm_out)

model = keras.Model(inputs=inputs, outputs=outputs)
model.compile(optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
model.summary()
```

Model: "functional_1"

Layer (type)	Output Shape	Par
input_layer_1 (InputLayer)	(None, 120, 7)	
lstm_1 (LSTM)	(None, 32)	5
dense_1 (Dense)	(None, 1)	

Total params: 5,153 (20.13 KB)
Trainable params: 5,153 (20.13 KB)
Non-trainable params: 0 (0.00 B)

5) Train your model:

Specify the file path where the model's weights will be saved with: `path_checkpoint = "model_checkpoint.weights.h5"`

We want to add a callback to stop training when a monitored metric stops improving: `es_callback = keras.callbacks.EarlyStopping(monitor="val_loss", min_delta=0, patience=5)`

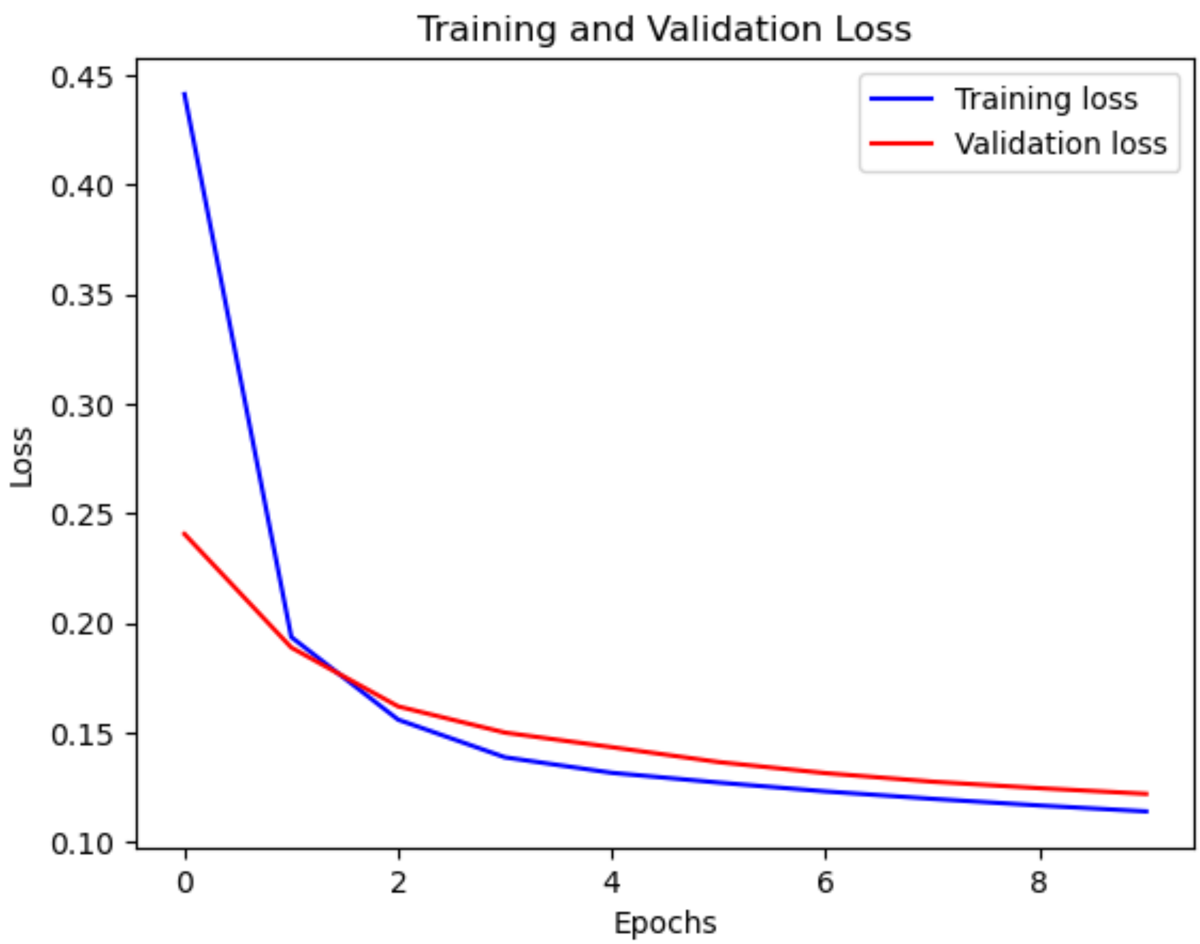
Train the model using Fit

In [34]:
```
path_checkpoint = "model_checkpoint.weights.h5"
es_callback = keras.callbacks.EarlyStopping(monitor="val_loss", min_delta=0,

modelckpt_callback = keras.callbacks.ModelCheckpoint(
    monitor="val_loss",
    filepath=path_checkpoint,
    verbose=1,
    save_weights_only=True,
    save_best_only=True,
)

history = model.fit(
    dataset_train,
    epochs=epochs,
    validation_data=dataset_val,
    callbacks=[es_callback, modelckpt_callback],
)
```

```
Epoch 1/10
95/96 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.8377
Epoch 1: val_loss improved from inf to 0.24050, saving model to model_checkp
oint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - loss: 0.8295 - val_loss: 0.2405
Epoch 2/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.2062
Epoch 2: val_loss improved from 0.24050 to 0.18863, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 3s 36ms/step - loss: 0.2058 - val_loss: 0.1886
Epoch 3/10
95/96 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step - loss: 0.1589
Epoch 3: val_loss improved from 0.18863 to 0.16174, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 38ms/step - loss: 0.1588 - val_loss: 0.1617
Epoch 4/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - loss: 0.1389
Epoch 4: val_loss improved from 0.16174 to 0.14978, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 39ms/step - loss: 0.1389 - val_loss: 0.1498
Epoch 5/10
95/96 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.1322
Epoch 5: val_loss improved from 0.14978 to 0.14314, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 36ms/step - loss: 0.1322 - val_loss: 0.1431
Epoch 6/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.1286
Epoch 6: val_loss improved from 0.14314 to 0.13633, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - loss: 0.1286 - val_loss: 0.1363
Epoch 7/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.1243
Epoch 7: val_loss improved from 0.13633 to 0.13128, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 37ms/step - loss: 0.1243 - val_loss: 0.1313
Epoch 8/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 0s 33ms/step - loss: 0.1212
Epoch 8: val_loss improved from 0.13128 to 0.12738, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 39ms/step - loss: 0.1212 - val_loss: 0.1274
Epoch 9/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - loss: 0.1186
Epoch 9: val_loss improved from 0.12738 to 0.12442, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 36ms/step - loss: 0.1185 - val_loss: 0.1244
Epoch 10/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step - loss: 0.1163
Epoch 10: val_loss improved from 0.12442 to 0.12182, saving model to model_c
heckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 4s 39ms/step - loss: 0.1163 - val_loss: 0.1218
```

Plot the results of your training:

In [35]:
```
def visualize_loss(history, title):
    loss = history.history["loss"]
    val_loss = history.history["val_loss"]
    epochs = range(len(loss))
    plt.figure()
    plt.plot(epochs, loss, "b", label="Training loss")
    plt.plot(epochs, val_loss, "r", label="Validation loss")
    plt.title(title)
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()


visualize_loss(history, "Training and Validation Loss")
```

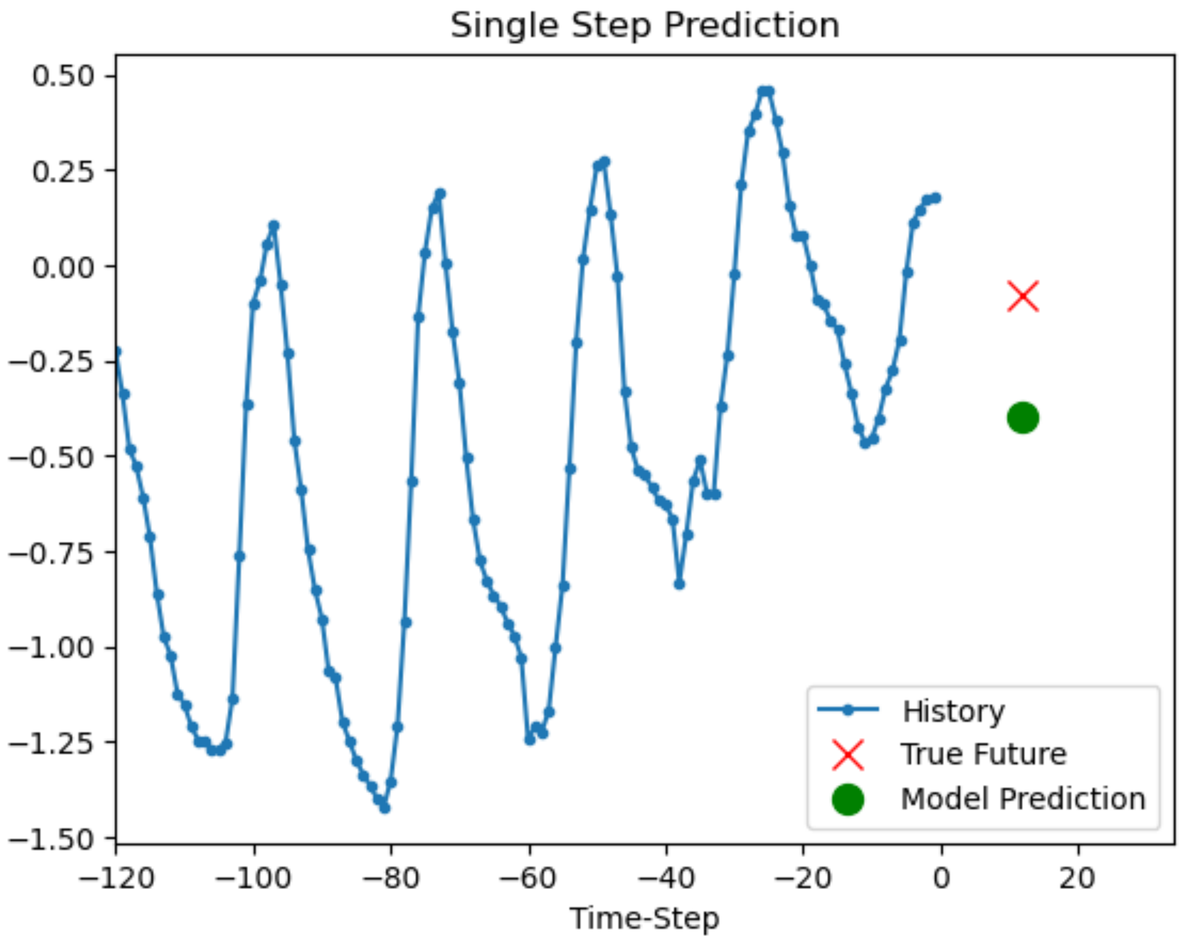


Make 5 predictions and display the predicted value

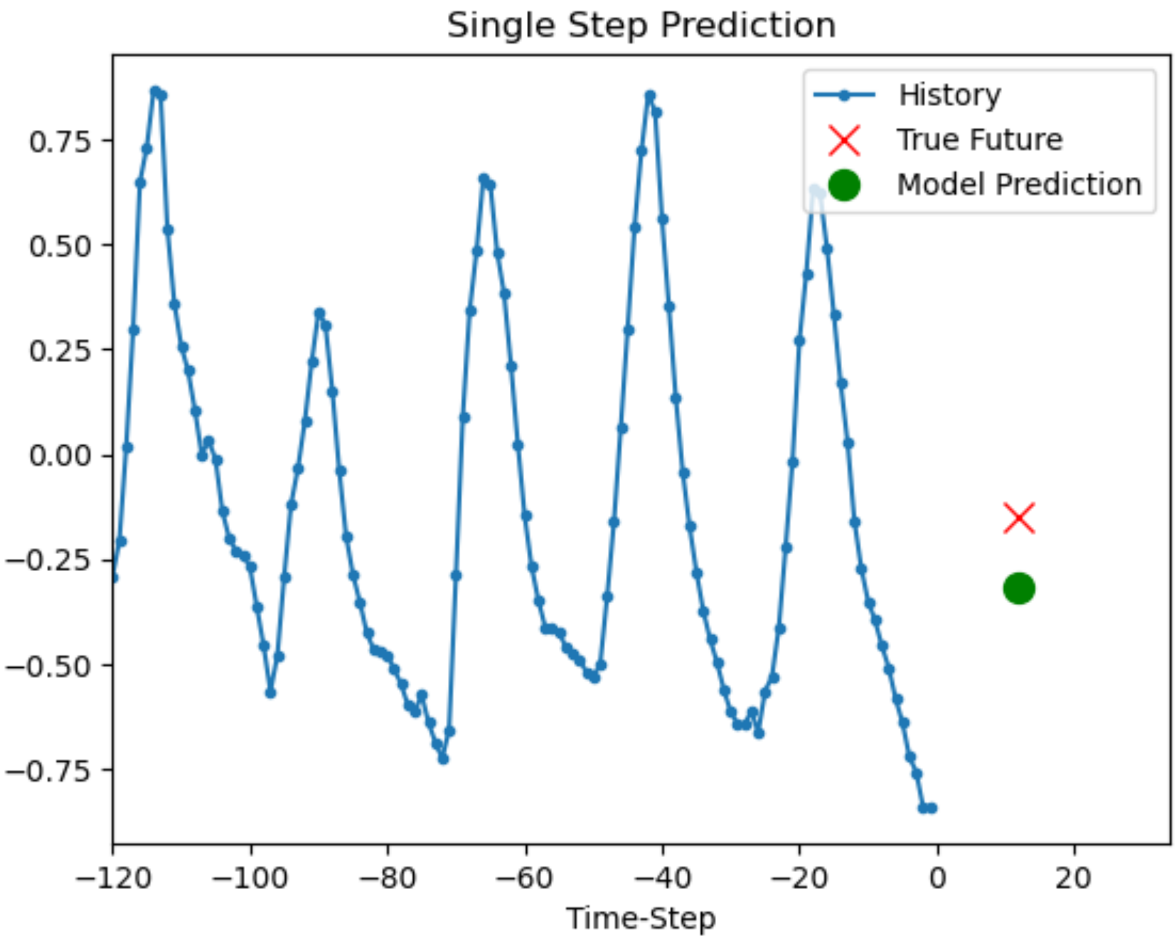
In [36]:
```
def show_plot(plot_data, delta, title):
```

```
    labels = ["History", "True Future", "Model Prediction"]
    marker = [".-", "rx", "go"]
    time_steps = list(range(-(plot_data[0].shape[0]), 0))
    if delta:
        future = delta
    else:
        future = 0

    plt.title(title)
    for i, val in enumerate(plot_data):
        if i:
            plt.plot(future, plot_data[i], marker[i], markersize=10, label=l
        else:
            plt.plot(time_steps, plot_data[i].flatten(), marker[i], label=la
    plt.legend()
    plt.xlim([time_steps[0], (future + 5) * 2])
    plt.xlabel("Time-Step")
    plt.show()
    return


for x, y in dataset_val.take(5):
    show_plot(
        [x[0][:, 1].numpy(), y[0].numpy(), model.predict(x)[0]],
        12,
        "Single Step Prediction",
    )
```

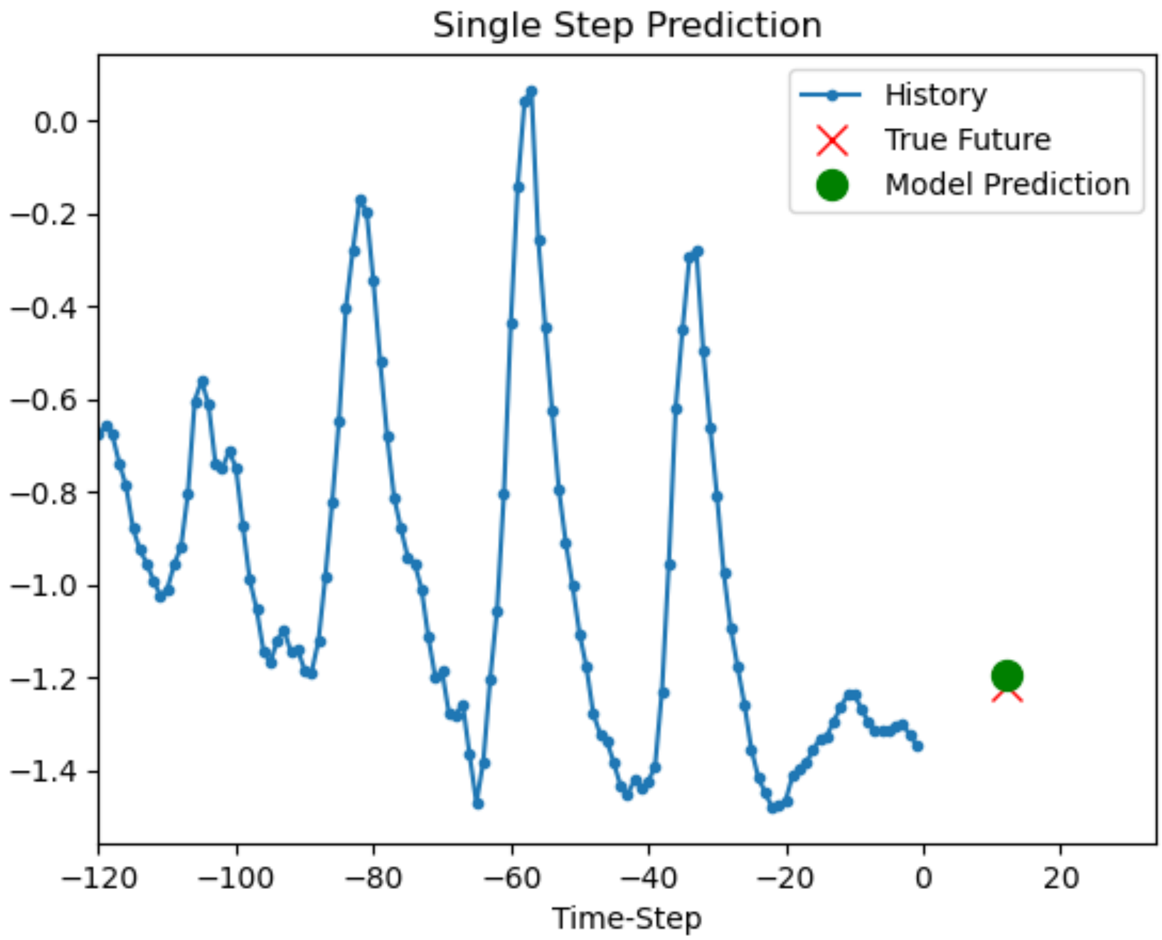
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```



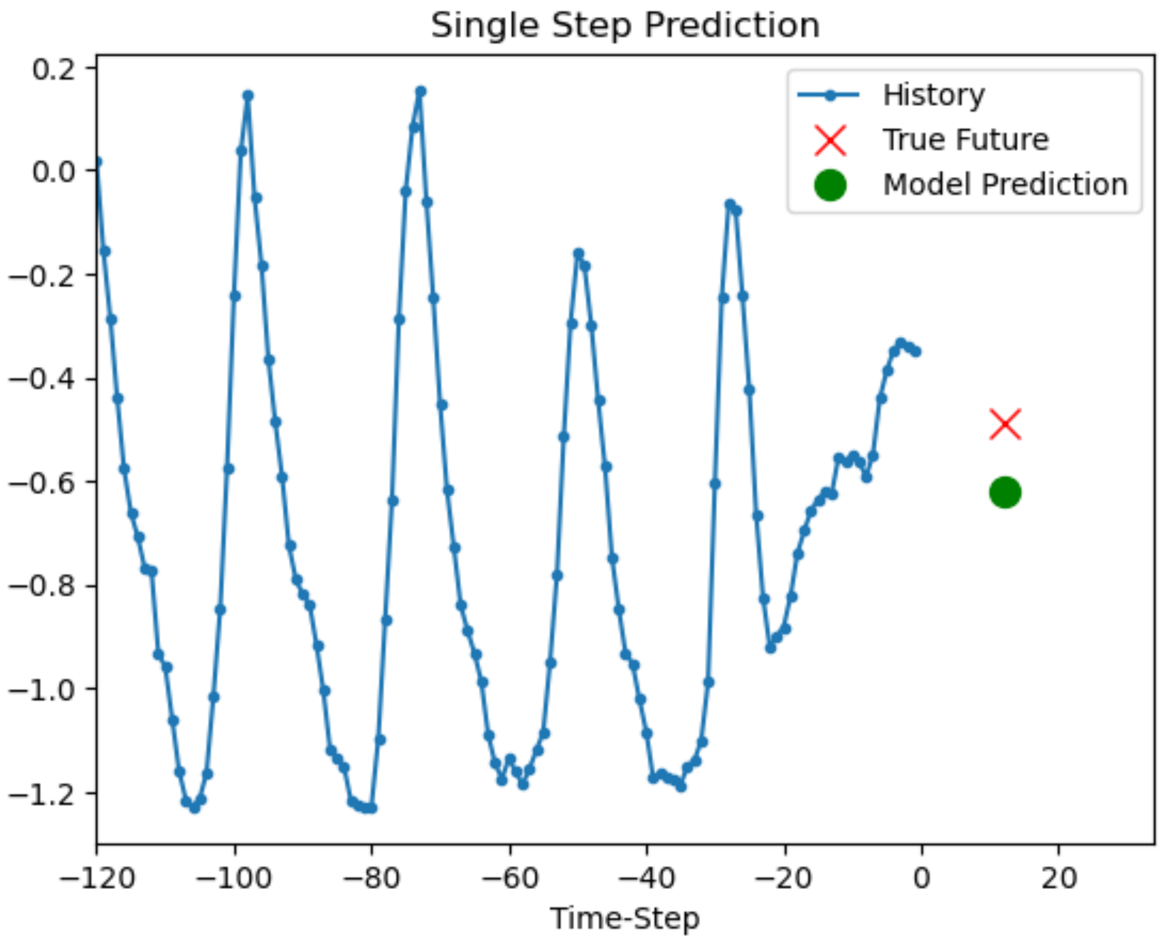

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```



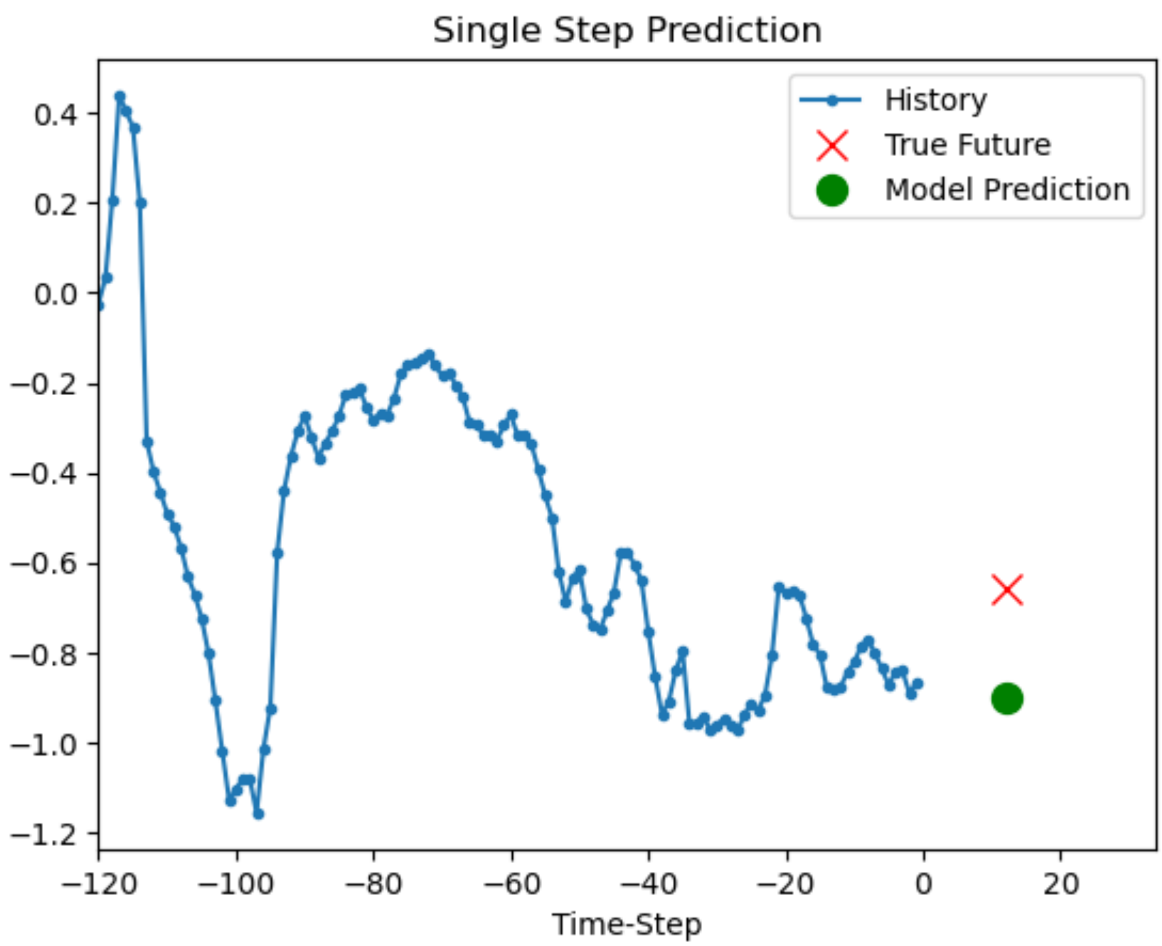
8/8 ━━━━━━━━━━━━━━━━━━━━ **0s** 3ms/step




```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```



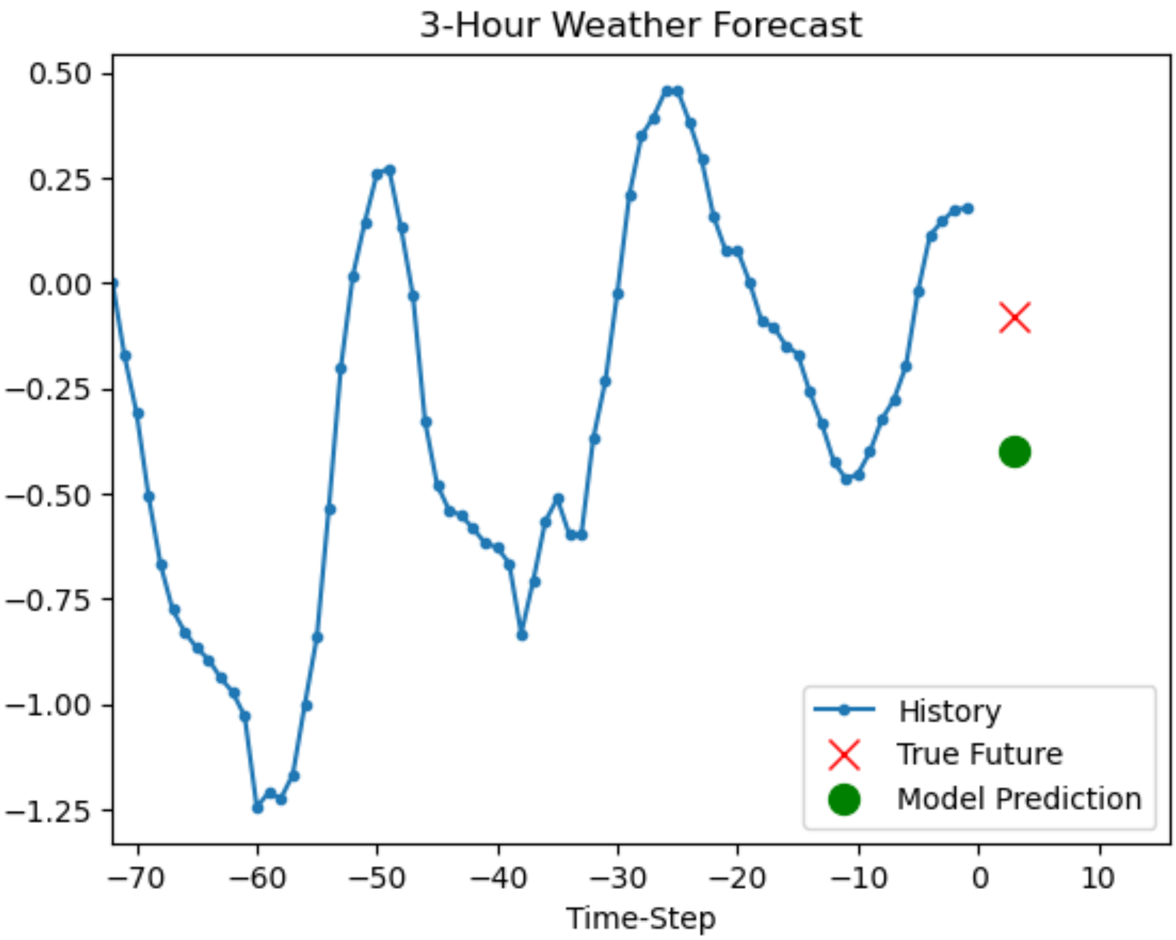

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```



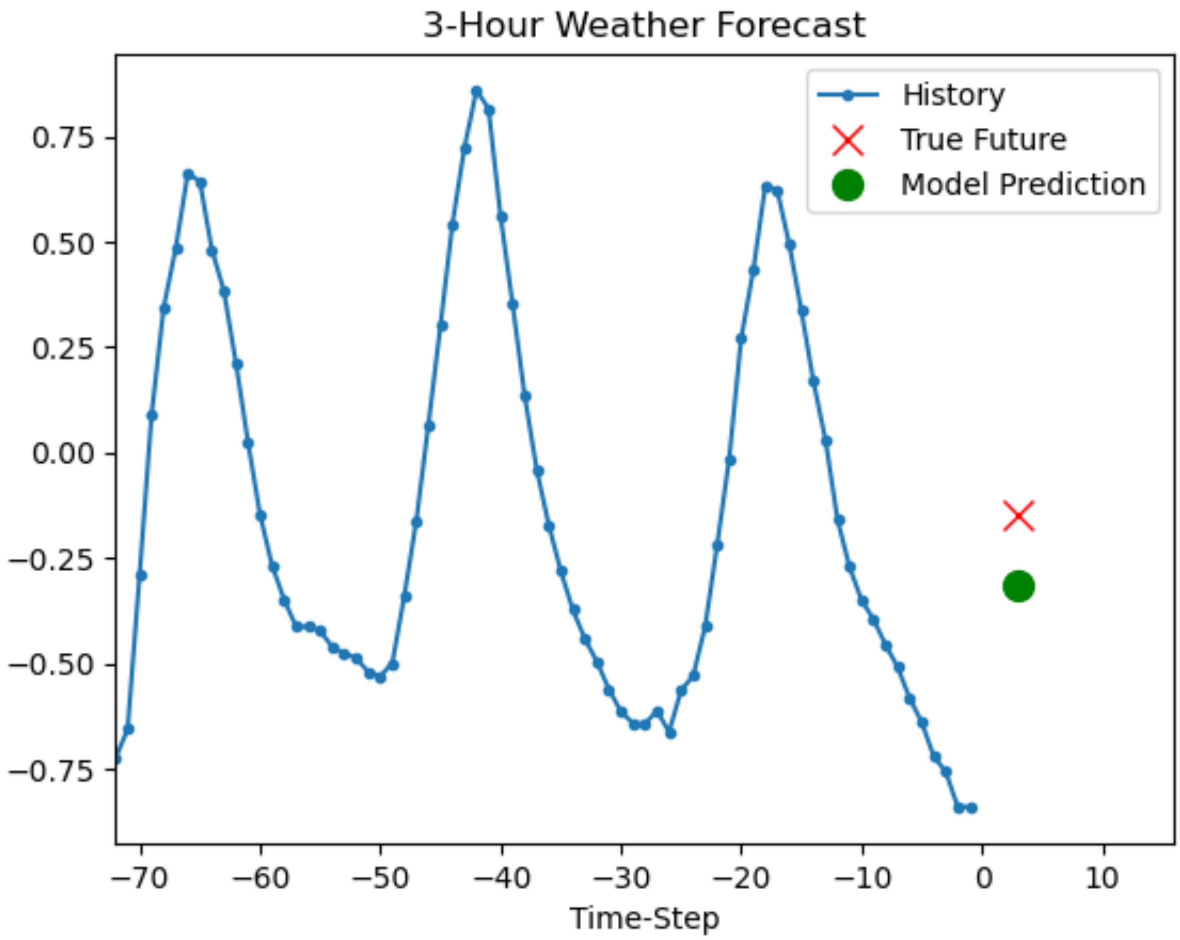
Now make a Time Series Forecasting where using the last 3 days you will predict the weather in the next 3 hours.

In [38]:
```
# Function to predict the next 3 hours of weather data using the last 3 days
def forecast_next_hours(model, dataset, past_days=3, future_hours=3):
    # Assuming data points are recorded every hour and the dataset is prepro
    past_steps = past_days * 24  # Convert days to hours
    future_steps = future_hours  # Future time horizon in hours

    for x, y in dataset.take(5):  # Taking one batch from the dataset for pr
        past_data = x[0][:, 1].numpy()[-past_steps:]  # Selecting the last 3
        predicted_future = model.predict(x)[0][:future_steps]  # Predicting

        # Visualize the forecast
        show_plot([past_data, y[0].numpy(), predicted_future], future_steps,

forecast_next_hours(model,dataset_val, 3, 3)
```

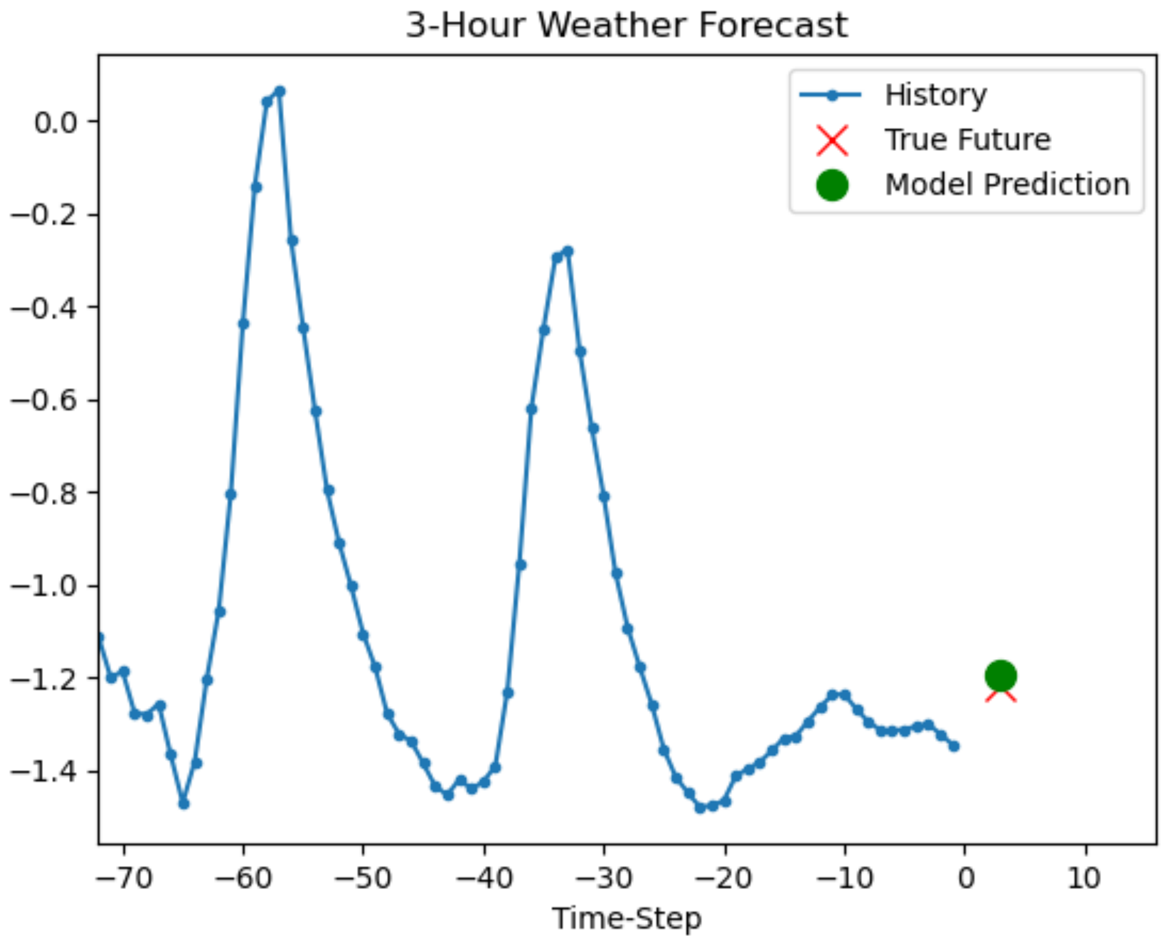
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```



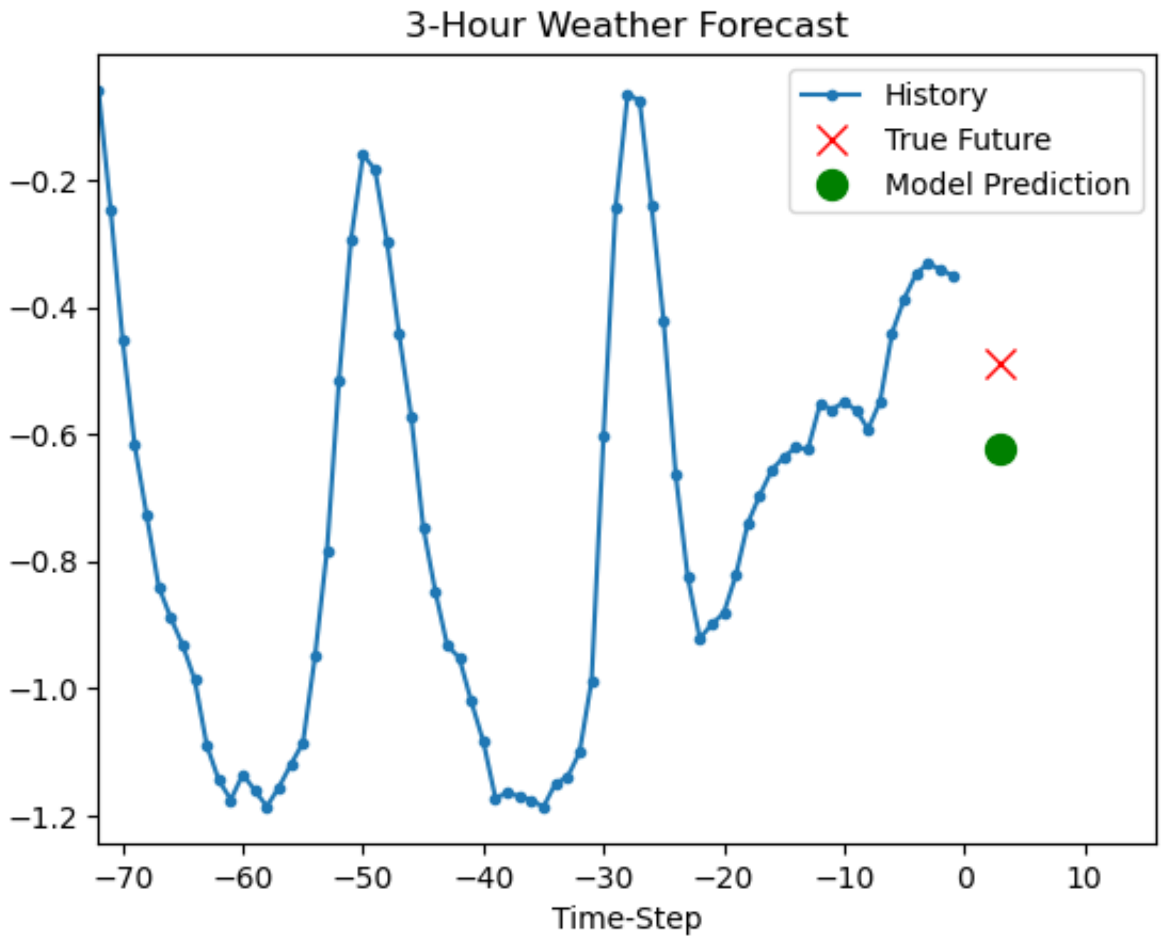

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```



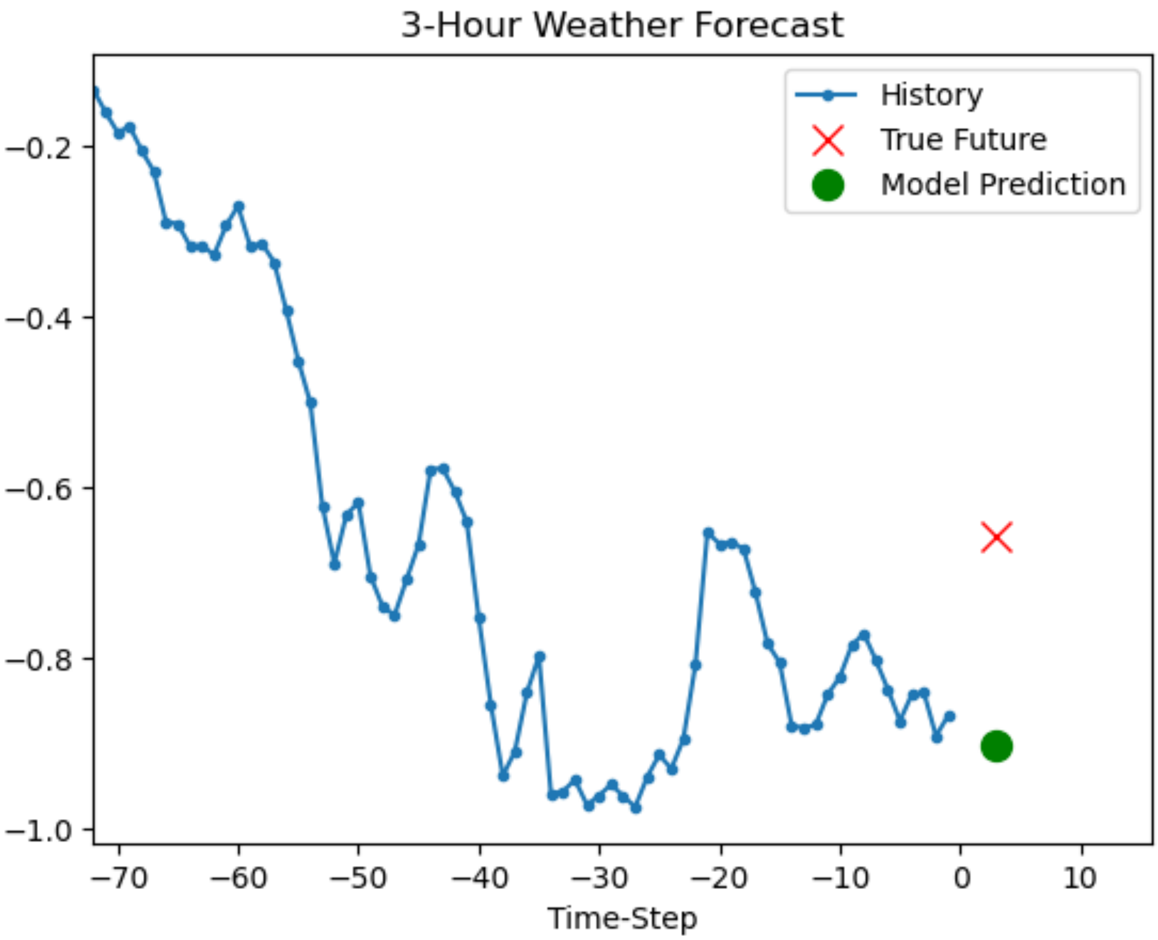

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```




```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
```




```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
```



In [40]:
```
past = 3*24
futre = 3

start = past + future
end = start + train_split

x_train = train_data[[i for i in range(7)]].values
y_train = features.iloc[start:end][[1]]

step = 1
sequence_length = past

dataset_train = keras.preprocessing.timeseries_dataset_from_array(
    x_train,
    y_train,
    sequence_length=sequence_length,
    sampling_rate=step,
    batch_size=batch_size,
)

x_end = len(val_data) - past - future
```

```
label_start = train_split + past + future

x_val = val_data.iloc[:x_end][[i for i in range(7)]].values
y_val = features.iloc[label_start:][[1]]

dataset_val = keras.preprocessing.timeseries_dataset_from_array(
    x_val,
    y_val,
    sequence_length=sequence_length,
    sampling_rate=step,
    batch_size=batch_size,
)


for batch in dataset_train.take(1):
    inputs, targets = batch

inputs = keras.layers.Input(shape=(inputs.shape[1], inputs.shape[2]))
lstm_out = keras.layers.LSTM(32)(inputs)
outputs = keras.layers.Dense(1)(lstm_out)

model2 = keras.Model(inputs=inputs, outputs=outputs)
model2.compile(optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
model2.summary()

path_checkpoint = "model_checkpoint.weights.h5"
es_callback = keras.callbacks.EarlyStopping(monitor="val_loss", min_delta=0,

modelckpt_callback = keras.callbacks.ModelCheckpoint(
    monitor="val_loss",
    filepath=path_checkpoint,
    verbose=1,
    save_weights_only=True,
    save_best_only=True,
)

history2 = model2.fit(
    dataset_train,
    epochs=epochs,
    validation_data=dataset_val,
    callbacks=[es_callback, modelckpt_callback],
)
```

```
2024-10-17 07:50:35.190696: I tensorflow/core/framework/local_rendezvous.cc:
404] Local rendezvous is aborting with status: OUT_OF_RANGE: End of sequence
```

Model: "functional_2"

Layer (type)	Output Shape	Par
input_layer_2 (InputLayer)	(None, 72, 7)	
lstm_2 (LSTM)	(None, 32)	5
dense_2 (Dense)	(None, 1)	

Total params: 5,153 (20.13 KB)

Trainable params: 5,153 (20.13 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
93/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.7858
Epoch 1: val_loss improved from inf to 0.21827, saving model to model_checkp
oint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 3s 24ms/step - loss: 0.7708 - val_loss: 0.2183
Epoch 2/10
93/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.1888
Epoch 2: val_loss improved from 0.21827 to 0.16519, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - loss: 0.1881 - val_loss: 0.1652
Epoch 3/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.1495
Epoch 3: val_loss improved from 0.16519 to 0.15137, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - loss: 0.1494 - val_loss: 0.1514
Epoch 4/10
93/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.1380
Epoch 4: val_loss improved from 0.15137 to 0.14586, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - loss: 0.1379 - val_loss: 0.1459
Epoch 5/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - loss: 0.1348
Epoch 5: val_loss improved from 0.14586 to 0.14186, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 23ms/step - loss: 0.1346 - val_loss: 0.1419
Epoch 6/10
95/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.1319
Epoch 6: val_loss improved from 0.14186 to 0.13757, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - loss: 0.1318 - val_loss: 0.1376
Epoch 7/10
94/96 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - loss: 0.1287
Epoch 7: val_loss improved from 0.13757 to 0.13381, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 25ms/step - loss: 0.1285 - val_loss: 0.1338
Epoch 8/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.1250
Epoch 8: val_loss improved from 0.13381 to 0.13096, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - loss: 0.1250 - val_loss: 0.1310
Epoch 9/10
95/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.1219
Epoch 9: val_loss improved from 0.13096 to 0.12863, saving model to model_ch
eckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - loss: 0.1218 - val_loss: 0.1286
Epoch 10/10
96/96 ━━━━━━━━━━━━━━━━━━━━ 0s 20ms/step - loss: 0.1190
Epoch 10: val_loss improved from 0.12863 to 0.12647, saving model to model_c
heckpoint.weights.h5
96/96 ━━━━━━━━━━━━━━━━━━━━ 2s 24ms/step - loss: 0.1189 - val_loss: 0.1265
```

In [47]:
```
df_loss_comparison = pd.DataFrame({
    'Model': ['Prediction 5', '3 Hour Prediction'],
    'Training Loss': [history.history['loss'][-1], history2.history['loss'][
    'Validation Loss': [history.history['val_loss'][-1], history2.history['v
})

# Mostrar la tabla
print(df_loss_comparison)
```

```
               Model  Training Loss  Validation Loss
0       Prediction 5       0.113810         0.121824
1  3 Hour Prediction       0.111597         0.126472
```

Notemos que los errores de validación y entrenamiento de ambas predicciones son casi iguales.