In [1]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Time series forecasting

[View on TensorFlow.org (https://www.tensorflow.org/tutorials/structured_data/time_series)]

[Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/structured_data/time_series.ipynb)]

[View source on GitHub (https://github.com/tensorflow/docs/blob/master/site/en/tutorials/structured_data/time_series.ipynb)]

[Download notebook (https://storage.googleapis.com/tensorflow_docs/docs/site/en/tutorials/structured_data/time_series.ipynb)]

This tutorial is an introduction to time series forecasting using Recurrent Neural Networks (RNNs). This is covered in two parts: first, you will forecast a univariate time series, then you will forecast a multivariate time series.

In [2]:
```
import tensorflow as tf

import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd

mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False
```

The weather dataset

This tutorial uses a [weather time series dataset (https://www.bgc-jena.mpg.de/wetter/)] recorded by the Max Planck Institute for Biogeochemistry (https://www.bgc-jena.mpg.de).

This dataset contains 14 different features such as air temperature, atmospheric pressure, and humidity. These were collected every 10 minutes, beginning in 2003. For efficiency, you will use only the data collected between 2009 and 2016. This section of the dataset was prepared by François Chollet for his book Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python).

In [3]:
```
zip_path = tf.keras.utils.get_file(
    origin='https://storage.googleapis.com/tensorflow/tf-keras-datasets/jena_climate_2009_2016.csv.zip',
    fname='jena_climate_2009_2016.csv.zip',
    extract=True)
csv_path, _ = os.path.splitext(zip_path)
```

In [4]:
```
df = pd.read_csv(csv_path)
```

Let's take a glance at the data.

In [5]:
```
df.head()
```

Out[5]:

	Date Time	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)
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	3.12	1307.75
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	3.03	1309.80
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	3.02	1310.24
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	3.08	1309.19
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	3.09	1309.00

As you can see above, an observation is recorded every 10 minutes. This means that, for a single hour, you will have 6 observations. Similarly, a single day will contain 144 (6x24) observations.

Given a specific time, let's say you want to predict the temperature 6 hours in the future. In order to make this prediction, you choose to use 5 days of observations. Thus, you would create a window containing the last 720(5x144) observations to train the model. Many such configurations are possible, making this dataset a good one to experiment with.

The function below returns the above described windows of time for the model to train on. The parameter `history_size` is the size of the past window of information. The `target_size` is how far in the future does the model need to learn to predict. The `target_size` is the label that needs to be predicted.

In [6]:
```
def univariate_data(dataset, start_index, end_index, history_size, target_size):
  data = []
  labels = []

  start_index = start_index + history_size
  if end_index is None:
    end_index = len(dataset) - target_size

  for i in range(start_index, end_index):
    indices = range(i-history_size, i)
    # Reshape data from (history_size,) to (history_size, 1)
    data.append(np.reshape(dataset[indices], (history_size, 1)))
    labels.append(dataset[i+target_size])
  return np.array(data), np.array(labels)
```

In both the following tutorials, the first 300,000 rows of the data will be the training dataset, and there remaining will be the validation dataset. This amounts to ~2100 days worth of training data.

In [7]:
```
TRAIN_SPLIT = 300000
```

Setting seed to ensure reproducibility.

In [8]:
```
tf.random.set_seed(13)
```

Part 1: Forecast a univariate time series

First, you will train a model using only a single feature (temperature), and use it to make predictions for that value in the future.

Let's first extract only the temperature from the dataset.

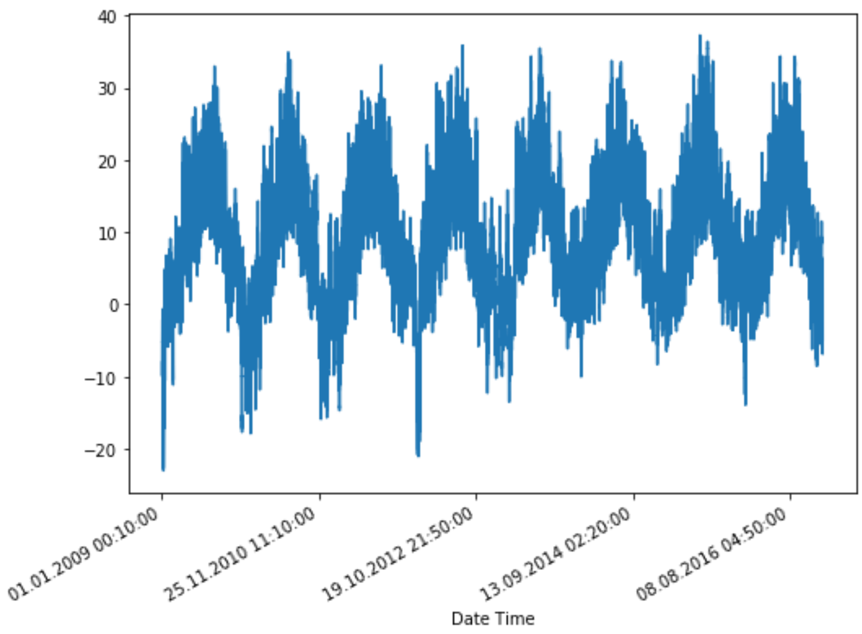
In [9]:
```
uni_data = df['T (degC)']
uni_data.index = df['Date Time']
uni_data.head()
```

Out[9]:
```
Date Time
01.01.2009 00:10:00   -8.02
01.01.2009 00:20:00   -8.41
01.01.2009 00:30:00   -8.51
01.01.2009 00:40:00   -8.31
01.01.2009 00:50:00   -8.27
Name: T (degC), dtype: float64
```

Let's observe how this data looks across time.

In [10]:
```
uni_data.plot(subplots=True)
```

Out[10]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x000001F30942BF48>],
      dtype=object)
```



In [11]:
```
uni_data = uni_data.values
```

It is important to scale features before training a neural network. Standardization is a common way of doing this scaling by subtracting the mean and dividing by the standard deviation of each feature.You could also use a `tf.keras.utils.normalize` method that rescales the values into a range of [0,1].

Note: The mean and standard deviation should only be computed using the training data.

In [12]:
```
uni_train_mean = uni_data[:TRAIN_SPLIT].mean()
uni_train_std = uni_data[:TRAIN_SPLIT].std()
```

Let's standardize the data.

In [13]:
```
uni_data = (uni_data-uni_train_mean)/uni_train_std
```

Let's now create the data for the univariate model. For part 1, the model will be given the last 20 recorded temperature observations, and needs to learn to predict the temperature at the next time step.

In [14]:
```
univariate_past_history = 20
univariate_future_target = 0

x_train_uni, y_train_uni = univariate_data(uni_data, 0, TRAIN_SPLIT,
                                           univariate_past_history,
                                           univariate_future_target)
x_val_uni, y_val_uni = univariate_data(uni_data, TRAIN_SPLIT, None,
                                       univariate_past_history,
                                       univariate_future_target)
```

This is what the `univariate_data` function returns.

In [15]:
```
print ('Single window of past history')
print (x_train_uni[0])
print ('\n Target temperature to predict')
print (y_train_uni[0])
```

```
Single window of past history
[[-1.99766294]
 [-2.04281897]
 [-2.05439744]
 [-2.0312405 ]
 [-2.02660912]
 [-2.00113649]
 [-1.95134907]
 [-1.95134907]
 [-1.98492663]
 [-2.04513467]
 [-2.08334362]
 [-2.09723778]
 [-2.09376424]
 [-2.09144854]
 [-2.07176515]
 [-2.07176515]
 [-2.07639653]
 [-2.08913285]
 [-2.09260639]
 [-2.10418486]]

 Target temperature to predict
-2.1041848598100876
```

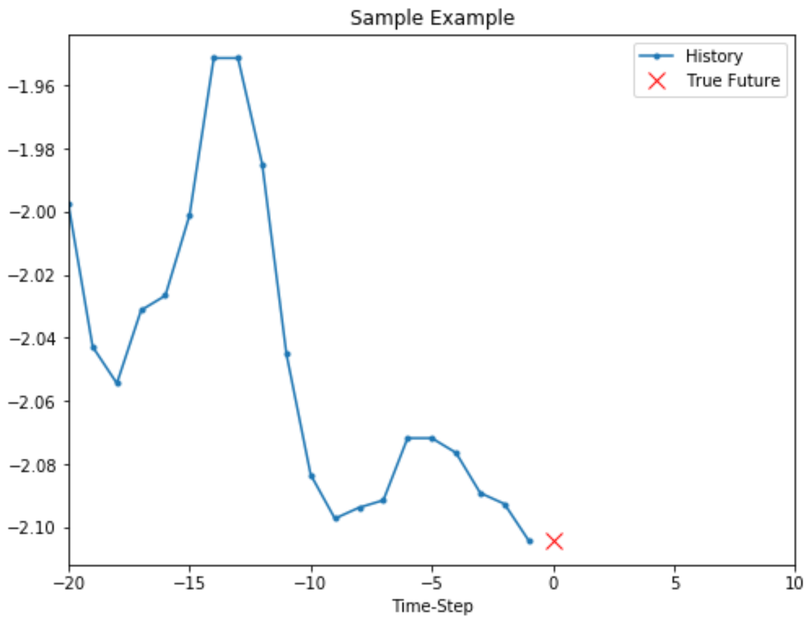
Now that the data has been created, let's take a look at a single example. The information given to the network is given in blue, and it must predict the value at the red cross.

In [16]:
```
def create_time_steps(length):
  return list(range(-length, 0))
```

In [17]:
```
def show_plot(plot_data, delta, title):
  labels = ['History', 'True Future', 'Model Prediction']
  marker = ['.-', 'rx', 'go']
  time_steps = create_time_steps(plot_data[0].shape[0])
  if delta:
    future = delta
  else:
    future = 0

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

In [18]:
```
show_plot([x_train_uni[0], y_train_uni[0]], 0, 'Sample Example')
```

Out[18]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\gcont\\Anaconda3\\envs\\tf\\li
b\\site-packages\\matplotlib\\pyplot.py'>
```



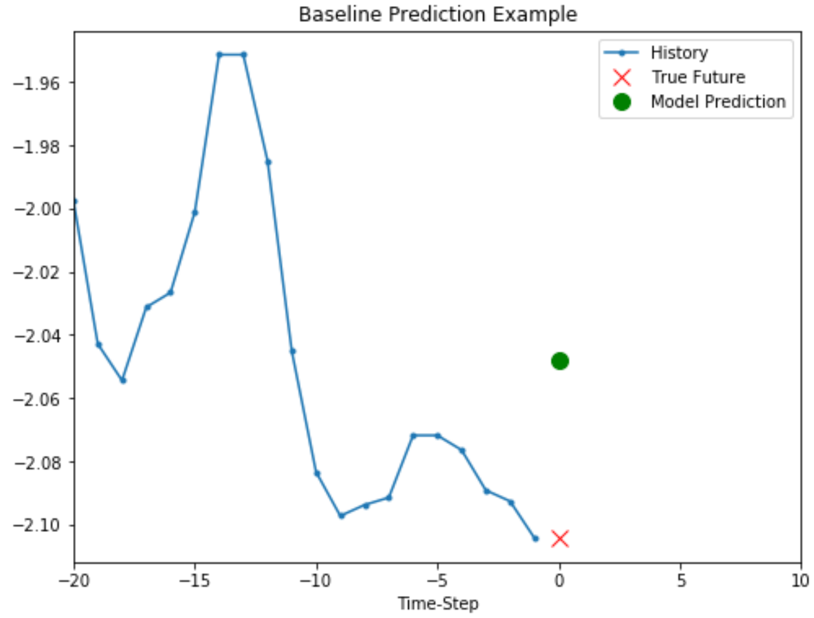
Baseline

Before proceeding to train a model, let's first set a simple baseline. Given an input point, the baseline method looks at all the history and predicts the next point to be the average of the last 20 observations.

In [19]:
```
def baseline(history):
  return np.mean(history)
```

In [20]:
```
show_plot([x_train_uni[0], y_train_uni[0], baseline(x_train_uni[0])], 0,
           'Baseline Prediction Example')
```

Out[20]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\gcont\\Anaconda3\\envs\\tf\\li
b\\site-packages\\matplotlib\\pyplot.py'>
```



Let's see if you can beat this baseline using a recurrent neural network.

Recurrent neural network

A Recurrent Neural Network (RNN) is a type of neural network well-suited to time series data. RNNs process a time series step-by-step, maintaining an internal state summarizing the information they've seen so far. For more details, read the RNN tutorial (https://www.tensorflow.org/tutorials/sequences/recurrent). In this tutorial, you will use a specialized RNN layer called Long Short Term Memory (LSTM (https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/LSTM))

Let's now use `tf.data` to shuffle, batch, and cache the dataset.

In [21]:
```
BATCH_SIZE = 256
BUFFER_SIZE = 10000

train_univariate = tf.data.Dataset.from_tensor_slices((x_train_uni, y_train_
uni))
train_univariate = train_univariate.cache().shuffle(BUFFER_SIZE).batch(BATCH
_SIZE).repeat()

val_univariate = tf.data.Dataset.from_tensor_slices((x_val_uni, y_val_uni))
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()
```

The following visualisation should help you understand how the data is represented after batching.

Time Series

You will see the LSTM requires the input shape of the data it is being given.

In [22]:
```
simple_lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(8, input_shape=x_train_uni.shape[-2:]),
    tf.keras.layers.Dense(1)
])

simple_lstm_model.compile(optimizer='adam', loss='mae')
```

Let's make a sample prediction, to check the output of the model.

In [23]:
```
for x, y in val_univariate.take(1):
    print(simple_lstm_model.predict(x).shape)
```

```
(256, 1)
```

Let's train the model now. Due to the large size of the dataset, in the interest of saving time, each epoch will only run for 200 steps, instead of the complete training data as normally done.

In [24]:
```
EVALUATION_INTERVAL = 200
EPOCHS = 10

simple_lstm_model.fit(train_univariate, epochs=EPOCHS,
                      steps_per_epoch=EVALUATION_INTERVAL,
                      validation_data=val_univariate, validation_steps=50)
```

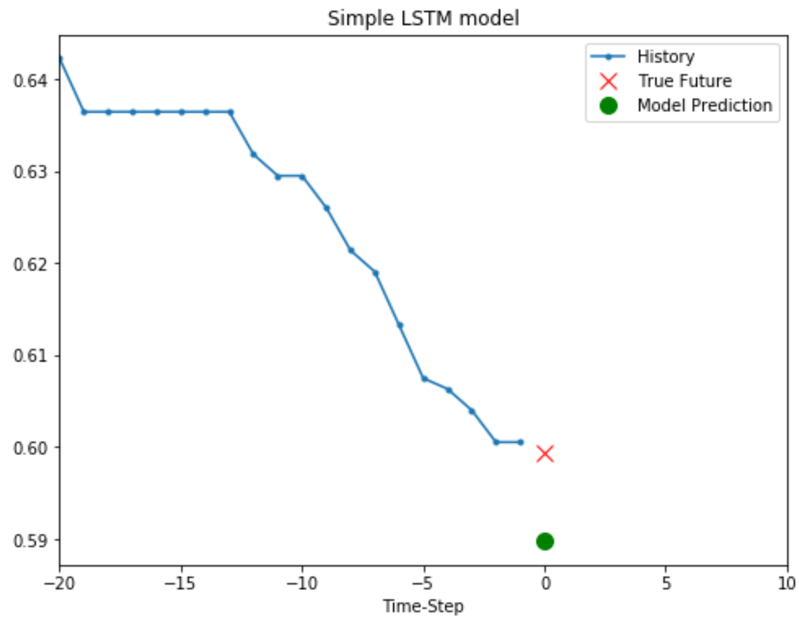
```
Train for 200 steps, validate for 50 steps
Epoch 1/10
200/200 [==============================] - 4s 20ms/step - loss: 0.4075 - val_
loss: 0.1351
Epoch 2/10
200/200 [==============================] - 2s 9ms/step - loss: 0.1118 - val_l
oss: 0.0359
Epoch 3/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0489 - val_l
oss: 0.0290
Epoch 4/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0443 - val_l
oss: 0.0258
Epoch 5/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0299 - val_l
oss: 0.0235
Epoch 6/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0317 - val_l
oss: 0.0224
Epoch 7/10
200/200 [==============================] - 2s 9ms/step - loss: 0.0286 - val_l
oss: 0.0206
Epoch 8/10
200/200 [==============================] - 2s 10ms/step - loss: 0.0263 - val_
loss: 0.0197
Epoch 9/10
200/200 [==============================] - 2s 10ms/step - loss: 0.0253 - val_
loss: 0.0182
Epoch 10/10
200/200 [==============================] - 2s 10ms/step - loss: 0.0227 - val_
loss: 0.0174
```

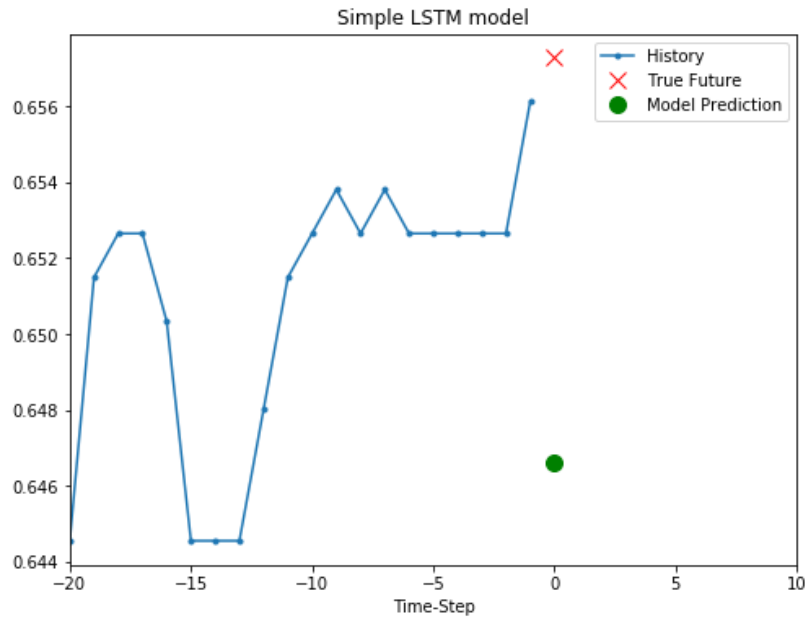
Out[24]:
```
<tensorflow.python.keras.callbacks.History at 0x1f30b08c788>
```

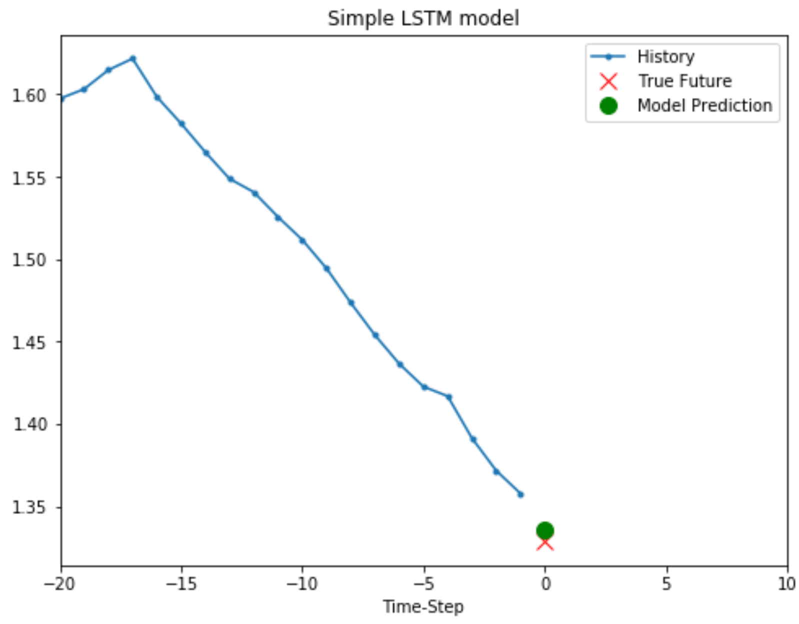
Predict using the simple LSTM model

Now that you have trained your simple LSTM, let's try and make a few predictions.

In [25]:
```
for x, y in val_univariate.take(3):
  plot = show_plot([x[0].numpy(), y[0].numpy(),
                    simple_lstm_model.predict(x)[0]], 0, 'Simple LSTM model
')
  plot.show()
```


Simple LSTM model
History
True Future
Model Prediction
0.64
0.63
0.62
0.61
0.60
0.59
−20
−15
−10
−5
0
5
10
Time-Step


Simple LSTM model
History
True Future
Model Prediction
0.656
0.654
0.652
0.650
0.648
0.646
0.644
−20
−15
−10
−5
0
5
10
Time-Step



This looks better than the baseline. Now that you have seen the basics, let's move on to part two, where you will work with a multivariate time series.

Part 2: Forecast a multivariate time series

The original dataset contains fourteen features. For simplicity, this section considers only three of the original fourteen. The features used are air temperature, atmospheric pressure, and air density.

To use more features, add their names to this list.

In [26]:
```
features_considered = ['p (mbar)', 'T (degC)', 'rho (g/m**3)']
```

In [27]:
```
features = df[features_considered]
features.index = df['Date Time']
features.head()
```

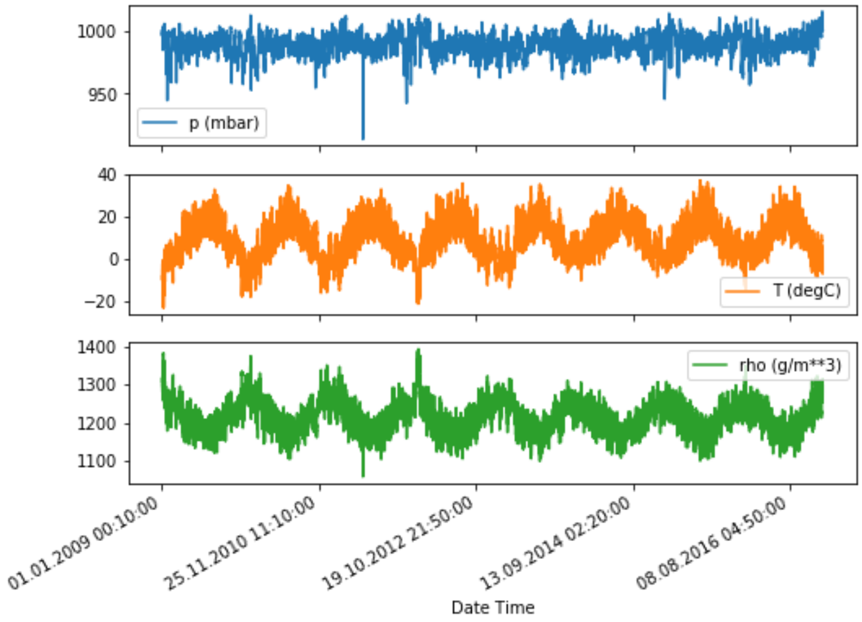
Out[27]:

Date Time	p (mbar)	T (degC)	rho (g/m**3)
01.01.2009 00:10:00	996.52	-8.02	1307.75
01.01.2009 00:20:00	996.57	-8.41	1309.80
01.01.2009 00:30:00	996.53	-8.51	1310.24
01.01.2009 00:40:00	996.51	-8.31	1309.19
01.01.2009 00:50:00	996.51	-8.27	1309.00

Let's have a look at how each of these features vary across time.

In [28]:
```
features.plot(subplots=True)
```

Out[28]:
```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x000001F313B7B508>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x000001F313B5C5C8>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x000001F313B7B408>],
      dtype=object)
```



As mentioned, the first step will be to standardize the dataset using the mean and standard deviation of the training data.

In [29]:
```
dataset = features.values
data_mean = dataset[:TRAIN_SPLIT].mean(axis=0)
data_std = dataset[:TRAIN_SPLIT].std(axis=0)
```

In [30]:
```
dataset = (dataset-data_mean)/data_std
```

Single step model

In a single step setup, the model learns to predict a single point in the future based on some history provided.

The below function performs the same windowing task as above, however, here it samples the past observation based on the step size given.

In [31]:
```
def multivariate_data(dataset, target, start_index, end_index, history_size,
                      target_size, step, single_step=False):
  data = []
  labels = []

  start_index = start_index + history_size
  if end_index is None:
    end_index = len(dataset) - target_size

  for i in range(start_index, end_index):
    indices = range(i-history_size, i, step)
    data.append(dataset[indices])

    if single_step:
      labels.append(target[i+target_size])
    else:
      labels.append(target[i:i+target_size])

  return np.array(data), np.array(labels)
```

In this tutorial, the network is shown data from the last five (5) days, i.e. 720 observations that are sampled every hour. The sampling is done every one hour since a drastic change is not expected within 60 minutes. Thus, 120 observation represent history of the last five days. For the single step prediction model, the label for a datapoint is the temperature 12 hours into the future. In order to create a label for this, the temperature after 72(12*6) observations is used.

In [32]:
```
past_history = 720
future_target = 72
STEP = 6

x_train_single, y_train_single = multivariate_data(dataset, dataset[:, 1],
0,
                                                   TRAIN_SPLIT, past_histor
y,
                                                   future_target, STEP,
                                                   single_step=True)
x_val_single, y_val_single = multivariate_data(dataset, dataset[:, 1],
                                               TRAIN_SPLIT, None, past_histo
ry,
                                               future_target, STEP,
                                               single_step=True)
```

Let's look at a single data-point.

In [33]:
```
print ('Single window of past history : {}'.format(x_train_single[0].shape))
```

```
Single window of past history : (120, 3)
```

In [34]:
```
train_data_single = tf.data.Dataset.from_tensor_slices((x_train_single, y_tr
ain_single))
train_data_single = train_data_single.cache().shuffle(BUFFER_SIZE).batch(BAT
CH_SIZE).repeat()

val_data_single = tf.data.Dataset.from_tensor_slices((x_val_single, y_val_si
ngle))
val_data_single = val_data_single.batch(BATCH_SIZE).repeat()
```

In [35]:
```
single_step_model = tf.keras.models.Sequential()
single_step_model.add(tf.keras.layers.LSTM(32,
                                           input_shape=x_train_single.shape
[-2:]))
single_step_model.add(tf.keras.layers.Dense(1))

single_step_model.compile(optimizer=tf.keras.optimizers.RMSprop(), loss='mae
')
```

Let's check out a sample prediction.

In [36]:
```
for x, y in val_data_single.take(1):
  print(single_step_model.predict(x).shape)
```

```
(256, 1)
```

In [37]:
```
single_step_history = single_step_model.fit(train_data_single, epochs=EPOCH
S,
                                            steps_per_epoch=EVALUATION_INTER
VAL,
                                            validation_data=val_data_single,
                                            validation_steps=50)
```

```
Train for 200 steps, validate for 50 steps
Epoch 1/10
200/200 [==============================] - 34s 169ms/step - loss: 0.3090 - va
l_loss: 0.2647
Epoch 2/10
200/200 [==============================] - 38s 192ms/step - loss: 0.2623 - va
l_loss: 0.2429
Epoch 3/10
200/200 [==============================] - 48s 242ms/step - loss: 0.2614 - va
l_loss: 0.2474
Epoch 4/10
200/200 [==============================] - 1031s 5s/step - loss: 0.2569 - val
_loss: 0.2448
Epoch 5/10
200/200 [==============================] - 797s 4s/step - loss: 0.2267 - val_
loss: 0.2344
Epoch 6/10
200/200 [==============================] - 48s 239ms/step - loss: 0.2415 - va
l_loss: 0.2668
Epoch 7/10
200/200 [==============================] - 48s 239ms/step - loss: 0.2416 - va
l_loss: 0.2566
Epoch 8/10
200/200 [==============================] - 48s 238ms/step - loss: 0.2410 - va
l_loss: 0.2387
Epoch 9/10
200/200 [==============================] - 47s 236ms/step - loss: 0.2452 - va
l_loss: 0.2478
Epoch 10/10
200/200 [==============================] - 54s 272ms/step - loss: 0.2387 - va
l_loss: 0.2425
```

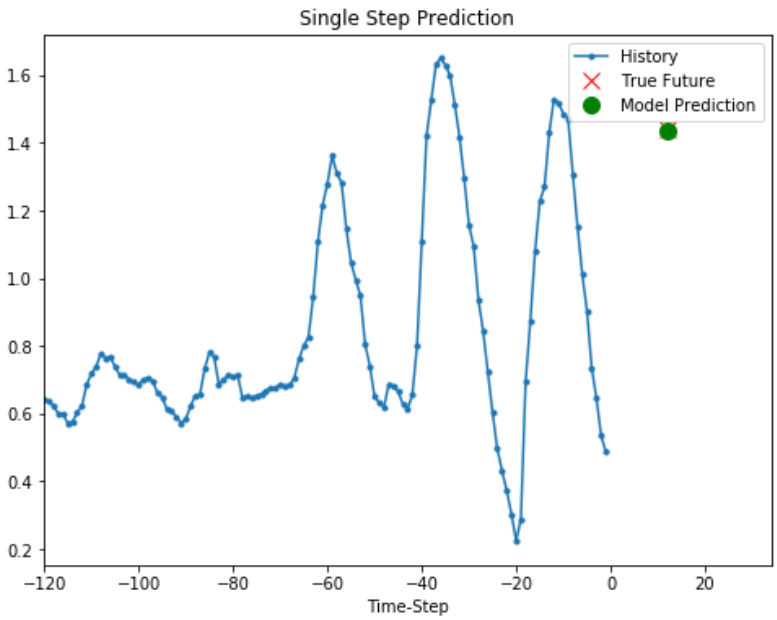
In [38]:
```
def plot_train_history(history, title):
  loss = history.history['loss']
  val_loss = history.history['val_loss']

  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.legend()

  plt.show()
```

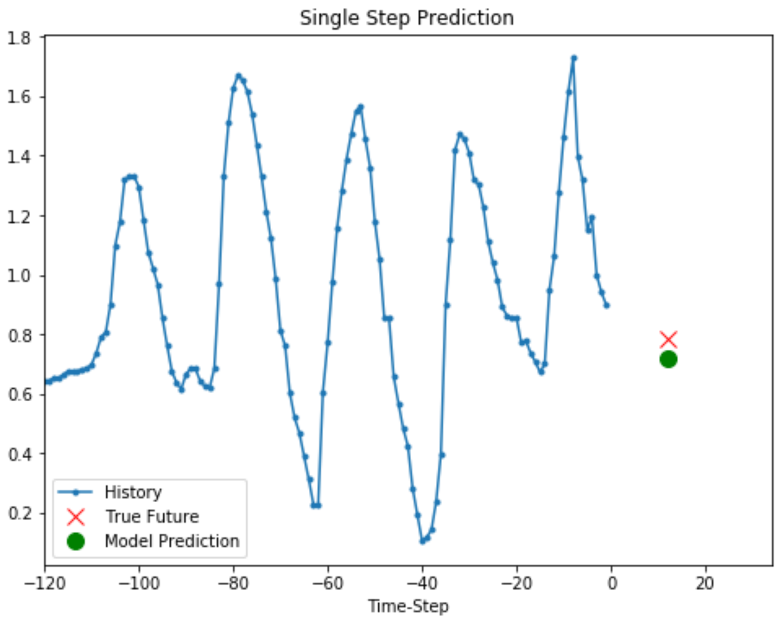
In [39]:
```
plot_train_history(single_step_history,
                   'Single Step Training and validation loss')
```

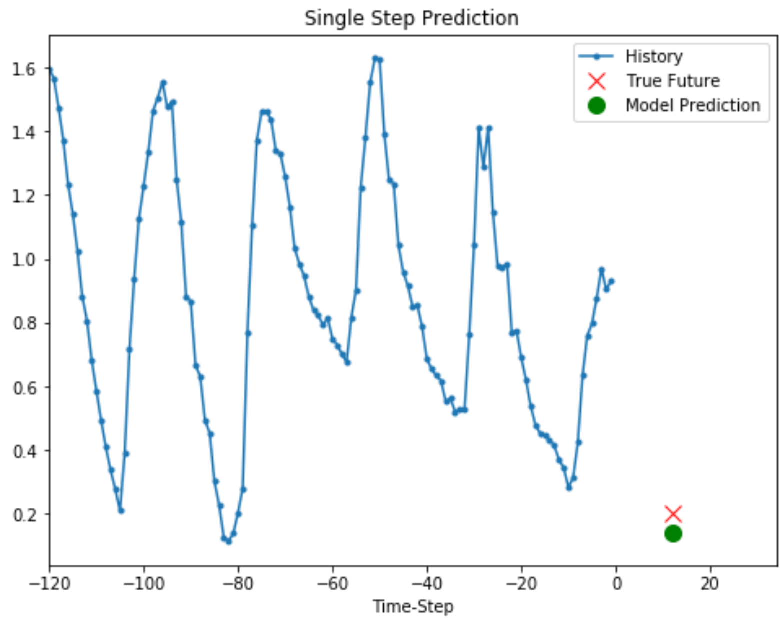
Predict a single step future

Now that the model is trained, let's make a few sample predictions. The model is given the history of three features over the past five days sampled every hour (120 data-points), since the goal is to predict the temperature, the plot only displays the past temperature. The prediction is made one day into the future (hence the gap between the history and prediction).

In [40]:
```
for x, y in val_data_single.take(3):
  plot = show_plot([x[0][:, 1].numpy(), y[0].numpy(),
                    single_step_model.predict(x)[0]], 12,
                   'Single Step Prediction')
  plot.show()
```


Single Step Prediction
History
True Future
Model Prediction
Time-Step


Single Step Prediction
History
True Future
Model Prediction
Time-Step



Multi-Step model

In a multi-step prediction model, given a past history, the model needs to learn to predict a range of future values. Thus, unlike a single step model, where only a single future point is predicted, a multi-step model predict a sequence of the future.

For the multi-step model, the training data again consists of recordings over the past five days sampled every hour. However, here, the model needs to learn to predict the temperature for the next 12 hours. Since an obversation is taken every 10 minutes, the output is 72 predictions. For this task, the dataset needs to be prepared accordingly, thus the first step is just to create it again, but with a different target window.

In [41]:
```
future_target = 72
x_train_multi, y_train_multi = multivariate_data(dataset, dataset[:, 1], 0,
                                                 TRAIN_SPLIT, past_history,
                                                 future_target, STEP)
x_val_multi, y_val_multi = multivariate_data(dataset, dataset[:, 1],
                                             TRAIN_SPLIT, None, past_histor
y,
                                             future_target, STEP)
```

Let's check out a sample data-point.

In [42]:
```
print ('Single window of past history : {}'.format(x_train_multi[0].shape))
print ('\n Target temperature to predict : {}'.format(y_train_multi[0].shap
e))
```

```
Single window of past history : (120, 3)

 Target temperature to predict : (72,)
```

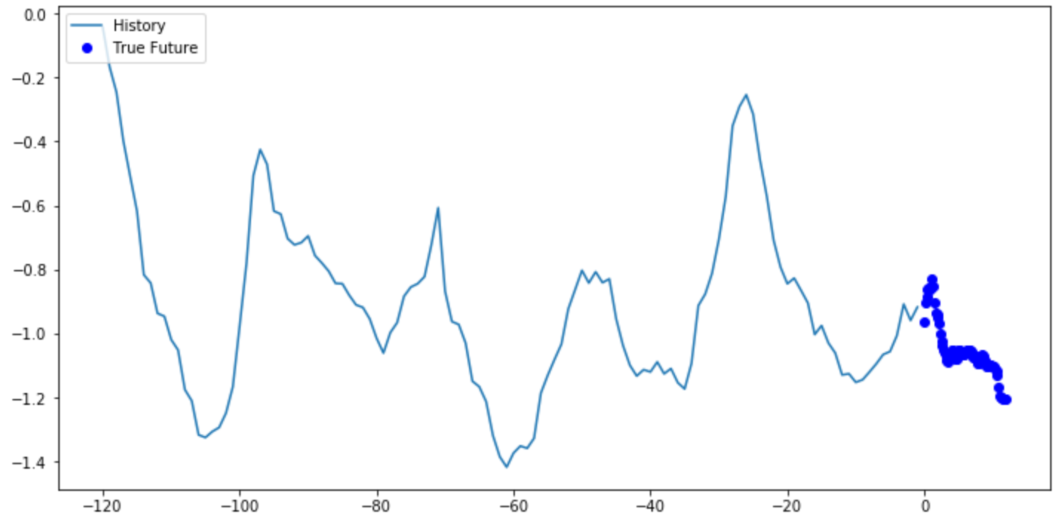
In [43]:
```
train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_multi, y_trai
n_multi))
train_data_multi = train_data_multi.cache().shuffle(BUFFER_SIZE).batch(BATCH
_SIZE).repeat()

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_multi, y_val_mult
i))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()
```

Plotting a sample data-point.

In [44]:
```
def multi_step_plot(history, true_future, prediction):
  plt.figure(figsize=(12, 6))
  num_in = create_time_steps(len(history))
  num_out = len(true_future)

  plt.plot(num_in, np.array(history[:, 1]), label='History')
  plt.plot(np.arange(num_out)/STEP, np.array(true_future), 'bo',
           label='True Future')
  if prediction.any():
    plt.plot(np.arange(num_out)/STEP, np.array(prediction), 'ro',
             label='Predicted Future')
  plt.legend(loc='upper left')
  plt.show()
```

In this plot and subsequent similar plots, the history and the future data are sampled every hour.

In [45]:
```
for x, y in train_data_multi.take(1):
  multi_step_plot(x[0], y[0], np.array([0]))
```



Since the task here is a bit more complicated than the previous task, the model now consists of two LSTM layers. Finally, since 72 predictions are made, the dense layer outputs 72 predictions.

In [46]:
```
multi_step_model = tf.keras.models.Sequential()
multi_step_model.add(tf.keras.layers.LSTM(32,
                                          return_sequences=True,
                                          input_shape=x_train_multi.shape[-
2:]))
multi_step_model.add(tf.keras.layers.LSTM(16, activation='relu'))
multi_step_model.add(tf.keras.layers.Dense(72))

multi_step_model.compile(optimizer=tf.keras.optimizers.RMSprop(clipvalue=1.
0), loss='mae')
```

Let's see how the model predicts before it trains.

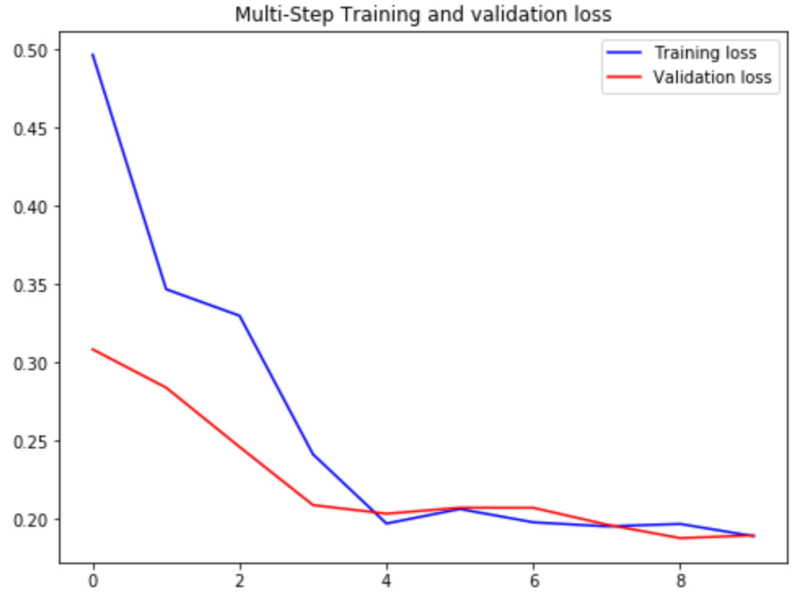
In [47]:
```
for x, y in val_data_multi.take(1):
  print (multi_step_model.predict(x).shape)
```

```
(256, 72)
```

In [48]:
```
multi_step_history = multi_step_model.fit(train_data_multi, epochs=EPOCHS,
                                          steps_per_epoch=EVALUATION_INTERVA
L,
                                          validation_data=val_data_multi,
                                          validation_steps=50)
```

```
Train for 200 steps, validate for 50 steps
Epoch 1/10
200/200 [==============================] - 87s 437ms/step - loss: 0.4969 - va
l_loss: 0.3084
Epoch 2/10
200/200 [==============================] - 108s 538ms/step - loss: 0.3469 - v
al_loss: 0.2840
Epoch 3/10
200/200 [==============================] - 130s 648ms/step - loss: 0.3299 - v
al_loss: 0.2460
Epoch 4/10
200/200 [==============================] - 122s 608ms/step - loss: 0.2413 - v
al_loss: 0.2088
Epoch 5/10
200/200 [==============================] - 133s 665ms/step - loss: 0.1971 - v
al_loss: 0.2034
Epoch 6/10
200/200 [==============================] - 159s 795ms/step - loss: 0.2061 - v
al_loss: 0.2072
Epoch 7/10
200/200 [==============================] - 174s 871ms/step - loss: 0.1978 - v
al_loss: 0.2071
Epoch 8/10
200/200 [==============================] - 177s 887ms/step - loss: 0.1953 - v
al_loss: 0.1964
Epoch 9/10
200/200 [==============================] - 171s 855ms/step - loss: 0.1968 - v
al_loss: 0.1877
Epoch 10/10
200/200 [==============================] - 164s 819ms/step - loss: 0.1890 - v
al_loss: 0.1895
```

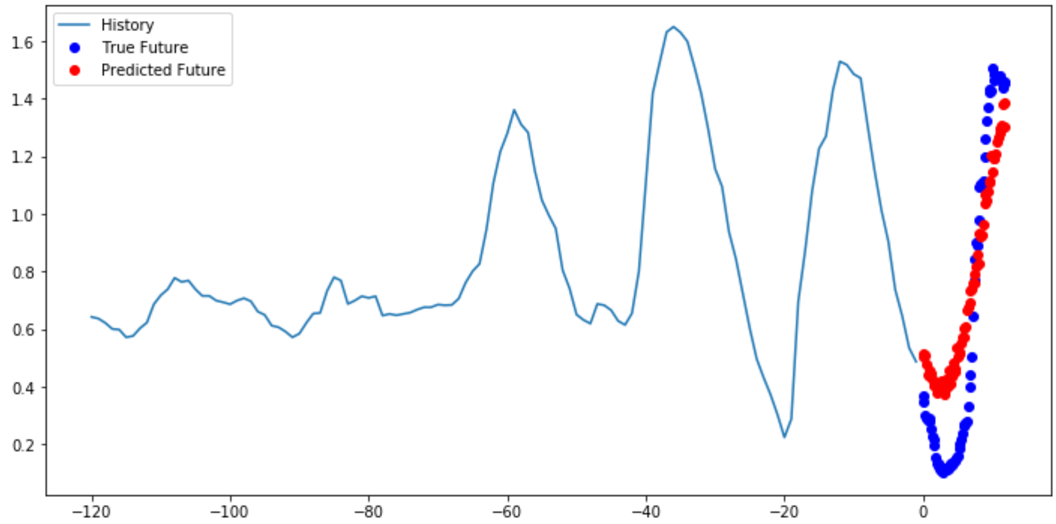
In [49]:
```
plot_train_history(multi_step_history, 'Multi-Step Training and validation l
oss')
```

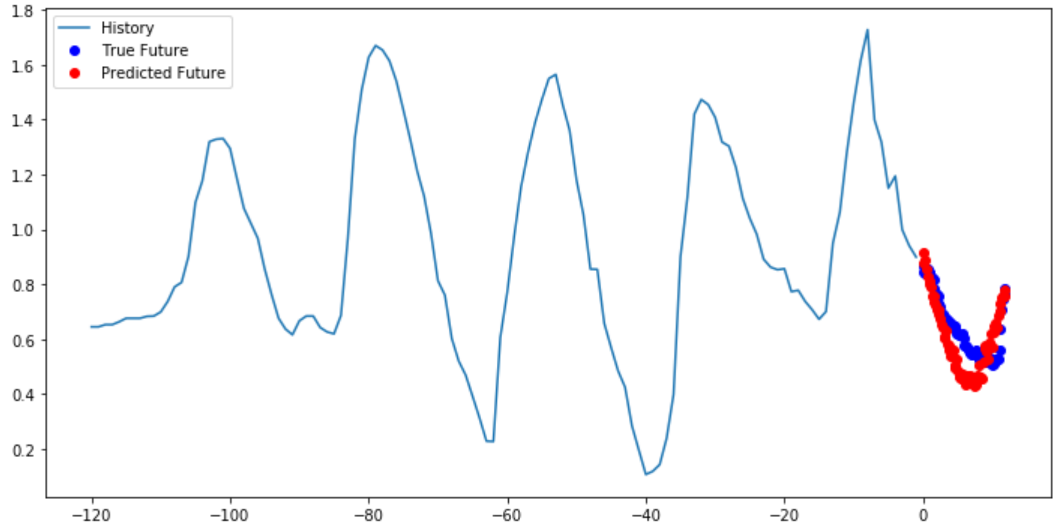


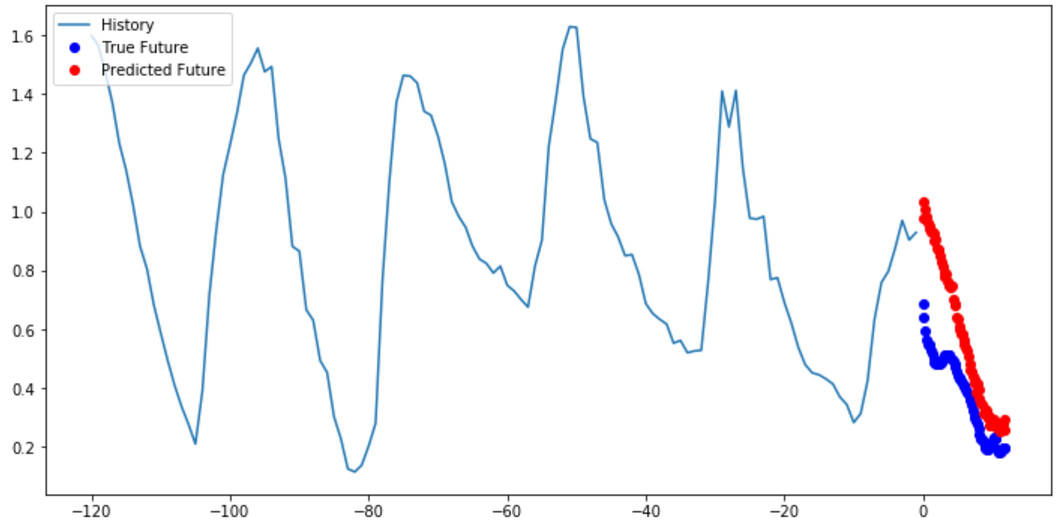
Predict a multi-step future

Let's now have a look at how well your network has learnt to predict the future.

In [50]:
```
for x, y in val_data_multi.take(3):
  multi_step_plot(x[0], y[0], multi_step_model.predict(x)[0])
```







Next steps

This tutorial was a quick introduction to time series forecasting using an RNN. You may now try to predict the stock market and become a billionaire.

In addition, you may also write a generator to yield data (instead of the uni/multivariate_data function), which would be more memory efficient. You may also check out this [time series windowing (https://www.tensorflow.org/guide/data#time_series_windowing)](https://www.tensorflow.org/guide/data#time_series_windowing) guide and use it in this tutorial.

For further understanding, you may read Chapter 15 of [Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow (https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/)](https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/), 2nd Edition and Chapter 6 of [Deep Learning with Python (https://www.manning.com/books/deep-learning-with-python)](https://www.manning.com/books/deep-learning-with-python).

In [ ]:

```
```