Deep Machine Learning - A6 - Time series forecasting

This notebook adapts the Tensorflow tutorial on Time series forecasting to data generated from a model for epidemic processes.

Quick disclaimer and clarification regarding this report

In this assignment, we were tasked with modifiying (extending) the provided jupyter notebook "Time series forecasting" that can be found HERE. In order to make this work we also need to load the data that can be found in the same repo, HERE. I did not make any changes to this raw data, instead i have made modifications to the models that are presented in this notebook.

In this notebook, chapters 1-6 is the exact copy of the provided code that was pulled from the git repo. I have highlighted my addition to this notebook in chapter 7, "MODIFIED MODELS". In here i will highlight all of the modifications and interpretations that i made so please check that out.

1. Imports and setup
2. Load and prepare the generated data
3. Baseline forecasting
4. Univariate LSTM based forecasting
5. Multivariate LSTM based forecasting - Single Step
6. Multivariate LSTM based forecasting - Multiple Steps
7. MODIFIED MODELS

Imports and setup

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
```

Load and prepare the generated data

We load data from the ODE model introduced in the notebook "Probability and Information Theory". For each of the 150 virtuel outbreaks (randomized and with different model parameters), we have time series (with 500 steps) for four the variables "Susceptible", "Infected", "Recovered", and "Deceased".

In [3]:
```
csv_path = "./epidemic_process_raw_data.csv"
df = pd.read_csv(csv_path)
df.head()
```

Out[3]:

	1	2	3	4	5	6	7	
0	100.287149	103.541223	95.879814	96.354848	96.980932	97.855310	98.940537	100.1
1	0.993774	1.017558	1.070030	1.116168	1.142078	1.134735	1.182418	1.2
2	0.000000	0.017741	0.036585	0.054735	0.074266	0.096065	0.117691	0.1
3	0.000000	0.000178	0.000364	0.000562	0.000757	0.000947	0.001160	0.0
4	103.489688	100.282780	96.634270	98.532514	99.089272	97.440900	98.416534	101.4

5 rows × 501 columns

In [4]:
```
dfSusceptible = df[df.index % 4 == 0]
dfSusceptible.head()
```

Out[4]:

	1	2	3	4	5	6	7	
0	100.287149	103.541223	95.879814	96.354848	96.980932	97.855310	98.940537	10
4	103.489688	100.282780	96.634270	98.532514	99.089272	97.440900	98.416534	10
8	101.527421	97.711732	96.168179	95.677962	95.575326	96.109792	96.943831	9
12	101.061107	99.112815	106.651686	101.622904	97.726686	95.692173	97.438263	10
16	101.957189	101.898022	100.881113	99.892000	98.939878	98.048565	98.220024	9

5 rows × 501 columns

In [5]:
```
dfInfected = df[df.index % 4 == 1]
dfInfected.head()
```

Out[5]:

	1	2	3	4	5	6	7	8	
1	0.993774	1.017558	1.070030	1.116168	1.142078	1.134735	1.182418	1.272310	1.35
5	1.021677	1.045410	1.120324	1.175914	1.236878	1.306676	1.387931	1.477973	1.54
9	1.020043	1.011238	1.031122	1.048642	1.049479	1.022891	1.035862	1.079177	1.11
13	1.035248	1.014189	1.133178	1.135622	1.157984	1.213088	1.281406	1.359858	1.42
17	1.012666	1.016949	1.053194	1.097599	1.143640	1.192369	1.238880	1.283688	1.32

5 rows × 501 columns

In [6]:

```
dfRecovered = df[df.index % 4 == 2]
dfRecovered.head()
```

Out[6]:

	1	2	3	4	5	6	7	8	9
2	0.0	0.017741	0.036585	0.054735	0.074266	0.096065	0.117691	0.139184	0.163615
6	0.0	0.017909	0.035748	0.056118	0.076620	0.097338	0.119592	0.143024	0.171253
10	0.0	0.016990	0.034644	0.052866	0.071444	0.090609	0.108733	0.126058	0.142408
14	0.0	0.017002	0.036315	0.057484	0.078381	0.098831	0.119563	0.140509	0.166486
18	0.0	0.017589	0.037434	0.056572	0.076275	0.096907	0.116533	0.135387	0.157592

5 rows × 501 columns

In [7]:

```
dfDead = df[df.index % 4 == 3]
dfDead.head()
```

Out[7]:

	1	2	3	4	5	6	7	8	9
3	0.0	0.000178	0.000364	0.000562	0.000757	0.000947	0.001160	0.001389	0.001635
7	0.0	0.000175	0.000351	0.000558	0.000763	0.000968	0.001196	0.001443	0.001719
11	0.0	0.000171	0.000352	0.000538	0.000729	0.000927	0.001126	0.001324	0.001488
15	0.0	0.000181	0.000364	0.000563	0.000774	0.001003	0.001192	0.001351	0.001575
19	0.0	0.000180	0.000358	0.000550	0.000740	0.000931	0.001138	0.001359	0.001574

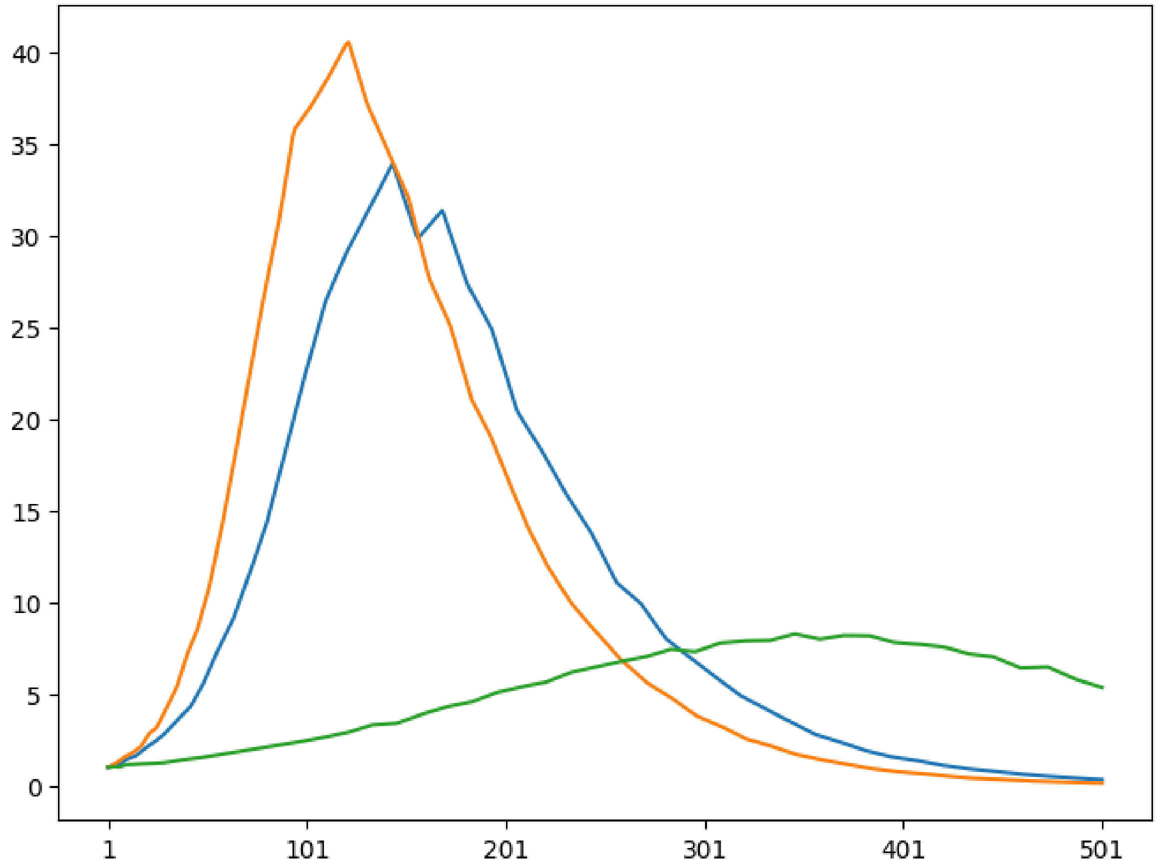
5 rows × 501 columns

Below a plot of three infection time series for the three first outbreaks.

In [8]:

```
dfInfected.loc[1,:].plot()
dfInfected.loc[5,:].plot()
dfInfected.loc[9,:].plot()
```

Out[8]: <Axes: >



We define a 90% / 10% of data for training / testing.

In [9]:
```
dfInfected_arr = dfInfected.values
dfInfected_arr.shape
TRAIN_SPLIT = int(dfInfected_arr.shape[0]-dfInfected_arr.shape[0]*0.1)
TRAIN_SPLIT
```

Out[9]: 135

We standardize the data.

In [10]:
```
uni_train_mean = dfInfected_arr[:TRAIN_SPLIT].mean()
uni_train_std = dfInfected_arr[:TRAIN_SPLIT].std()
uni_data = (dfInfected_arr-uni_train_mean)/uni_train_std
print ('\n Univariate data shape')
print(uni_data.shape)
```

```
 Univariate data shape
(150, 501)
```

We split the data into time series of `univariate_past_history=20` days length and predict the future of the current day, i.e., `univariate_future_target=0`, for the "infected" variable.

In [11]:
```
def univariate_data(dataset, start_series, end_series, history_size, target_size
    data = []
    labels = []
    start_index = history_size
    end_index = len(dataset[0]) - target_size
    for c in range(start_series, end_series):
        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[c][indices], (history_size, 1)))
            labels.append(dataset[c][i+target_size])
    return np.array(data), np.array(labels)
```

In [12]:
```
univariate_past_history = 20 #days
univariate_future_target = 0 #current day

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, len(uni_data),
                                        univariate_past_history,
                                        univariate_future_target)
```

In [13]:
```
print ('Single window of past history')
print (x_train_uni[0])
print ('\n Target number to predict')
print (y_train_uni[0])
print ('\n Number of traing data points')
print (y_train_uni.shape[0])
print ('\n Number of test data points')
print (x_val_uni.shape[0])
```

```
Single window of past history
[[-0.95291296]
 [-0.95044298]
 [-0.94499366]
 [-0.9402021 ]
 [-0.93751136]
 [-0.93827393]
 [-0.93332191]
 [-0.92398652]
 [-0.91523643]
 [-0.90667772]
 [-0.90243571]
 [-0.89846308]
 [-0.89449045]
 [-0.89051782]
 [-0.88593997]
 [-0.87701137]
 [-0.86808277]
 [-0.85915417]
 [-0.85022557]
 [-0.84167481]]

 Target number to predict
-0.8339932964893617

 Number of traing data points
64935

 Number of test data points
7215
```

In [14]:
```
def create_time_steps(length):
    return list(range(-length, 0))
```

In [15]:

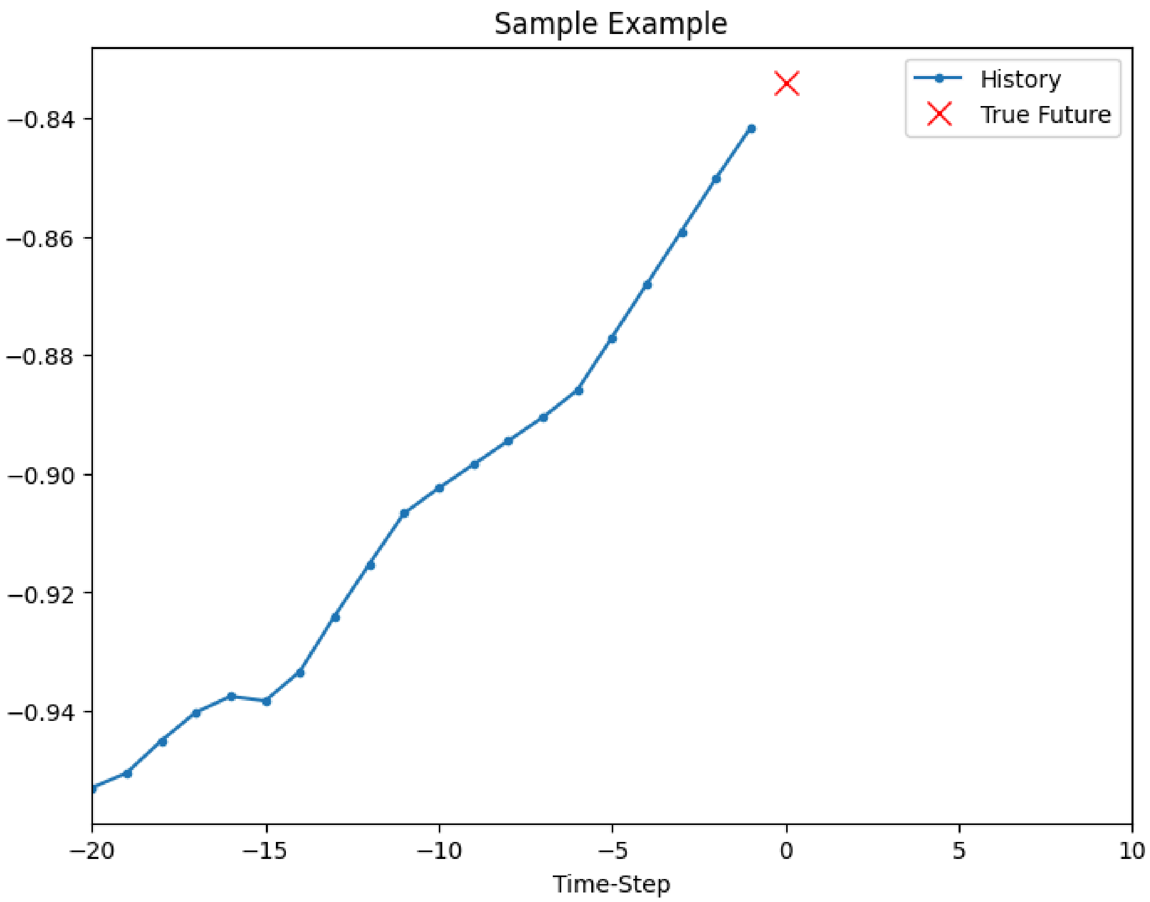
```
def show_plot(plot_data, delta, title):
    labels = ['History', 'True Future', 'Model Prediction']
    marker = ['.-', 'rx', 'go']
    time_steps = create_time_steps(plot_data[0].shape[0])
    if delta:
        future = delta
    else:
        future = 0
    plt.title(title)
    for i, x in enumerate(plot_data):
        if i:
            plt.plot(future, plot_data[i], marker[i], markersize=10,label=labels
        else:
            plt.plot(time_steps, plot_data[i].flatten(), marker[i], label=labels
    plt.legend()
    plt.xlim([time_steps[0], (future+5)*2])
    plt.xlabel('Time-Step')
    return plt
```

In [16]:

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

Out[16]:

```
<module 'matplotlib.pyplot' from 'C:\\Users\\kemal\\AppData\\Local\\Packages\\P
ythonSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\\LocalCache\\local-packages
\\Python310\\site-packages\\matplotlib\\pyplot.py'>
```



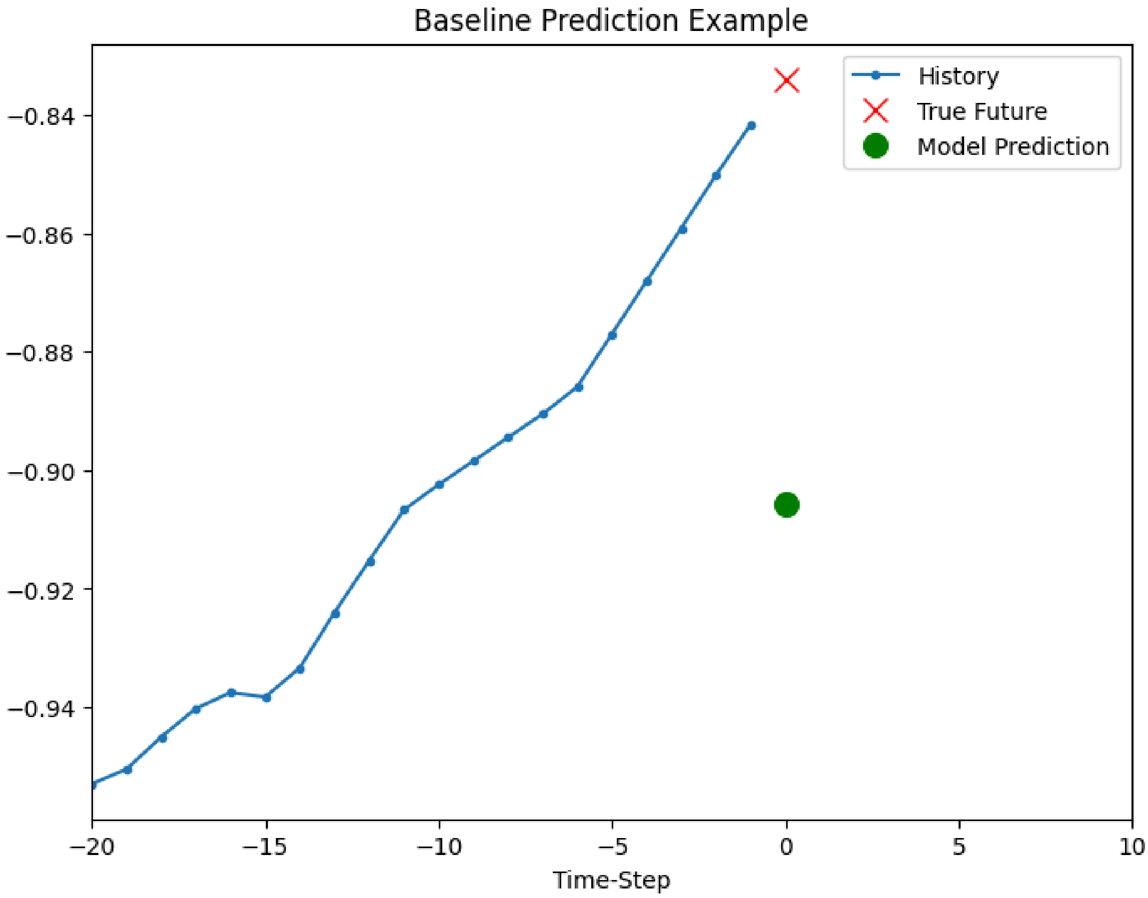
Baseline forecasting

Predicts the mean of the `history`.

In [17]:
```python
def baseline(history):
    return np.mean(history)
```

In [18]:
```python
show_plot([x_train_uni[0], y_train_uni[0], baseline(x_train_uni[0])], 0, 'Baseli
```

Out[18]:
```
<module 'matplotlib.pyplot' from 'C:\\Users\\kemal\\AppData\\Local\\Packages\\PythonSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\\LocalCache\\local-packages\\Python310\\site-packages\\matplotlib\\pyplot.py'>
```



Univariate LSTM based forecasting

In [19]:
```python
print (x_train_uni.shape)
print (y_train_uni.shape)
x_train_uni.dtype
```

```
(64935, 20, 1)
(64935,)
```

Out[19]:
```
dtype('float64')
```

Batching and resampling; the dataset is repeated indefinitely. Check the tutorial for the details.

In [20]:
```python
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_SIZ

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

```
val_univariate = val_univariate.batch(BATCH_SIZE).repeat()

train_univariate
```

Out[20]:
```
<_RepeatDataset element_spec=(TensorSpec(shape=(None, 20, 1), dtype=tf.float64, name=None), TensorSpec(shape=(None,), dtype=tf.float64, name=None))>
```

We define the first LSTM model with 8 units.

In [21]:
```
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')
simple_lstm_model.summary()
x_train_uni.shape[-2:]
```

```
C:\Users\kemal\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10_qbz5n2kfra8p0\LocalCache\local-packages\Python310\site-packages\keras\src\layers\rnn\rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 8)	320
dense (Dense)	(None, 1)	9

Total params: 329 (1.29 KB)

Trainable params: 329 (1.29 KB)

Non-trainable params: 0 (0.00 B)

Out[21]: (20, 1)

In [22]:
```
for x, y in val_univariate.take(1):
    print(simple_lstm_model.predict(x).shape)
    print(y.shape)
```

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
(256, 1)
(256,)
```

When passing an indefinitely repeated training data set, we need to specify the numbre of steps per training interval (epoch).

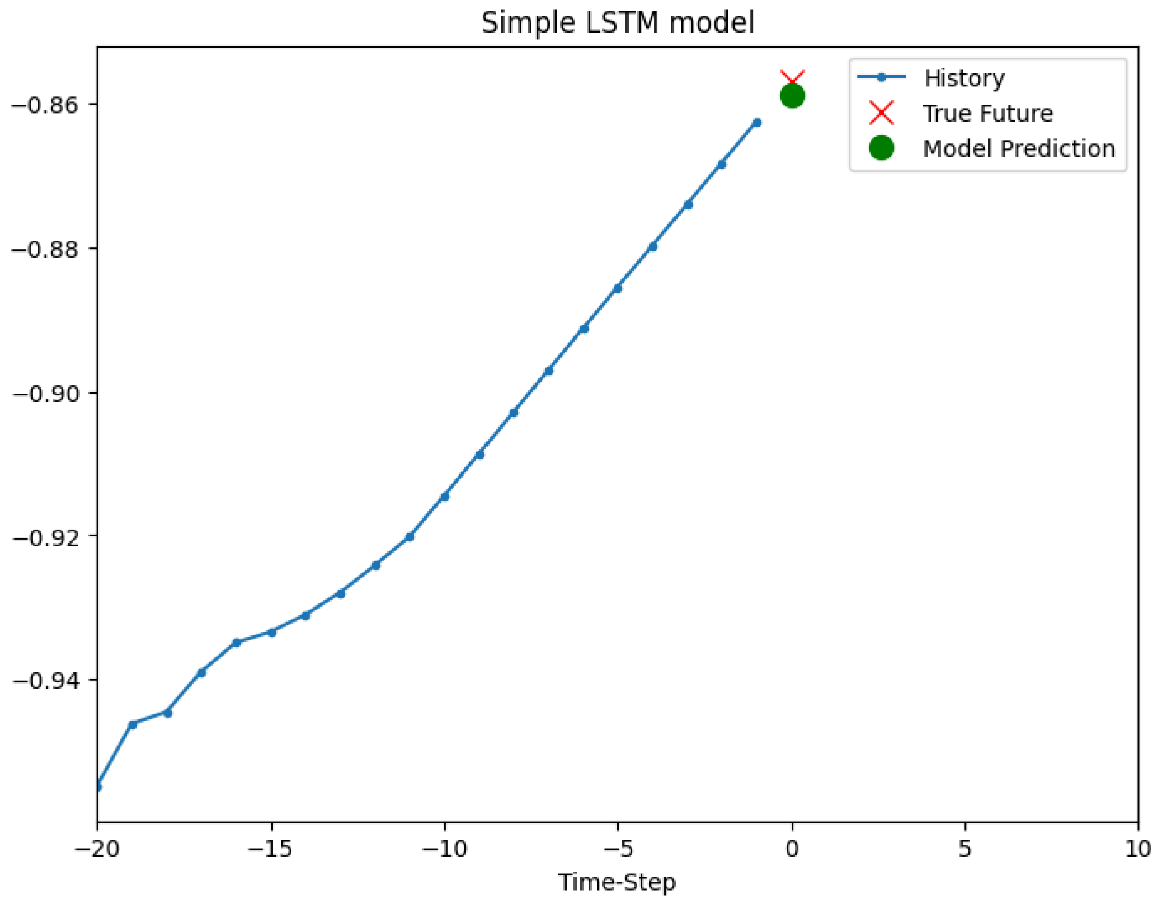
In [23]:
```
EVALUATION_INTERVAL = 2000
EPOCHS = 10

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

```
Epoch 1/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 15s 6ms/step - loss: 0.1108 - val_loss: 0.0042
Epoch 2/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - loss: 0.0048 - val_loss: 0.0029
Epoch 3/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - loss: 0.0034 - val_loss: 0.0024
Epoch 4/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - loss: 0.0029 - val_loss: 0.0022
Epoch 5/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - loss: 0.0026 - val_loss: 0.0020
Epoch 6/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - loss: 0.0024 - val_loss: 0.0020
Epoch 7/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 13s 6ms/step - loss: 0.0022 - val_loss: 0.0014
Epoch 8/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 12s 6ms/step - loss: 0.0021 - val_loss: 0.0016
Epoch 9/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - loss: 0.0020 - val_loss: 0.0020
Epoch 10/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - loss: 0.0020 - val_loss: 0.0020
```

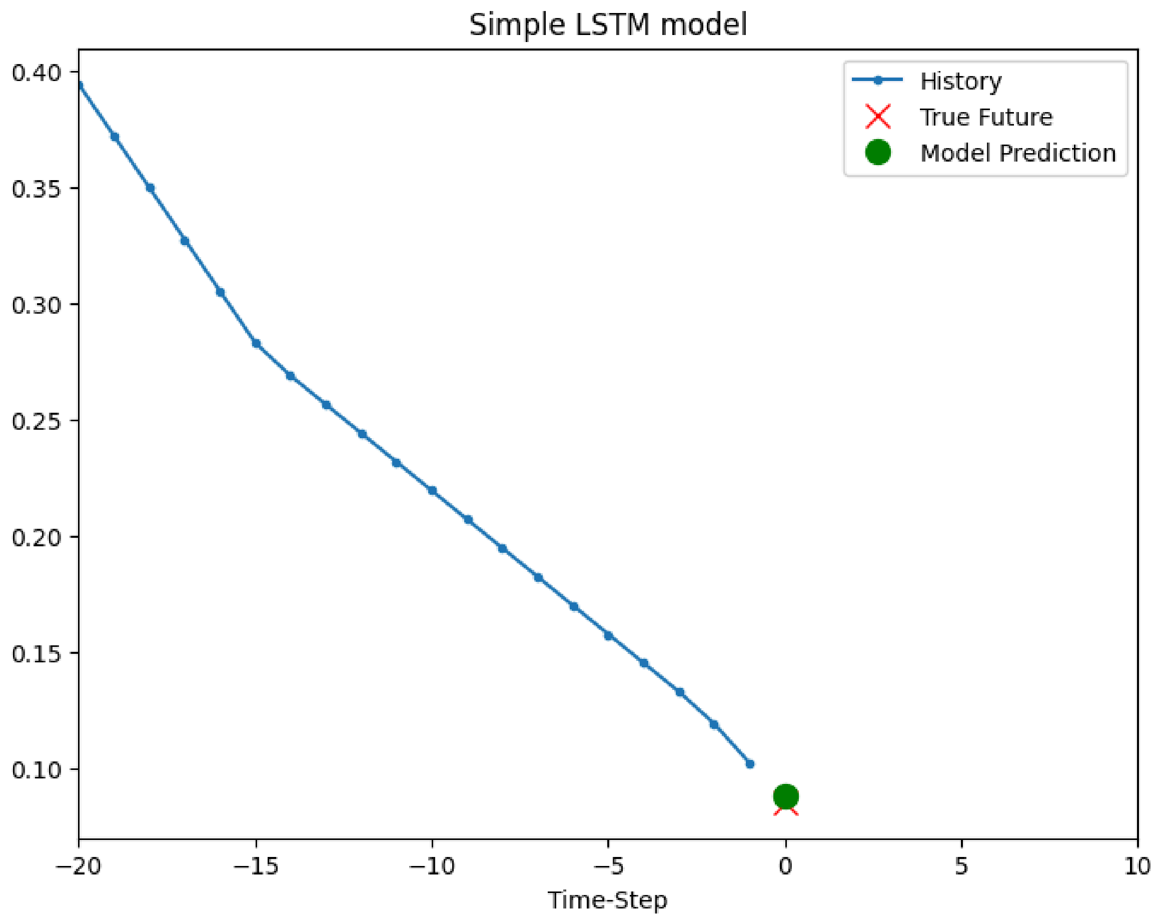
Out[23]: <keras.src.callbacks.history.History at 0x274c1ead180>

In [24]:
```
for x, y in val_univariate.take(3):
    plot = show_plot([x[0].numpy(), y[0].numpy(), simple_lstm_model.predict(x)[0
    plot.show()
```

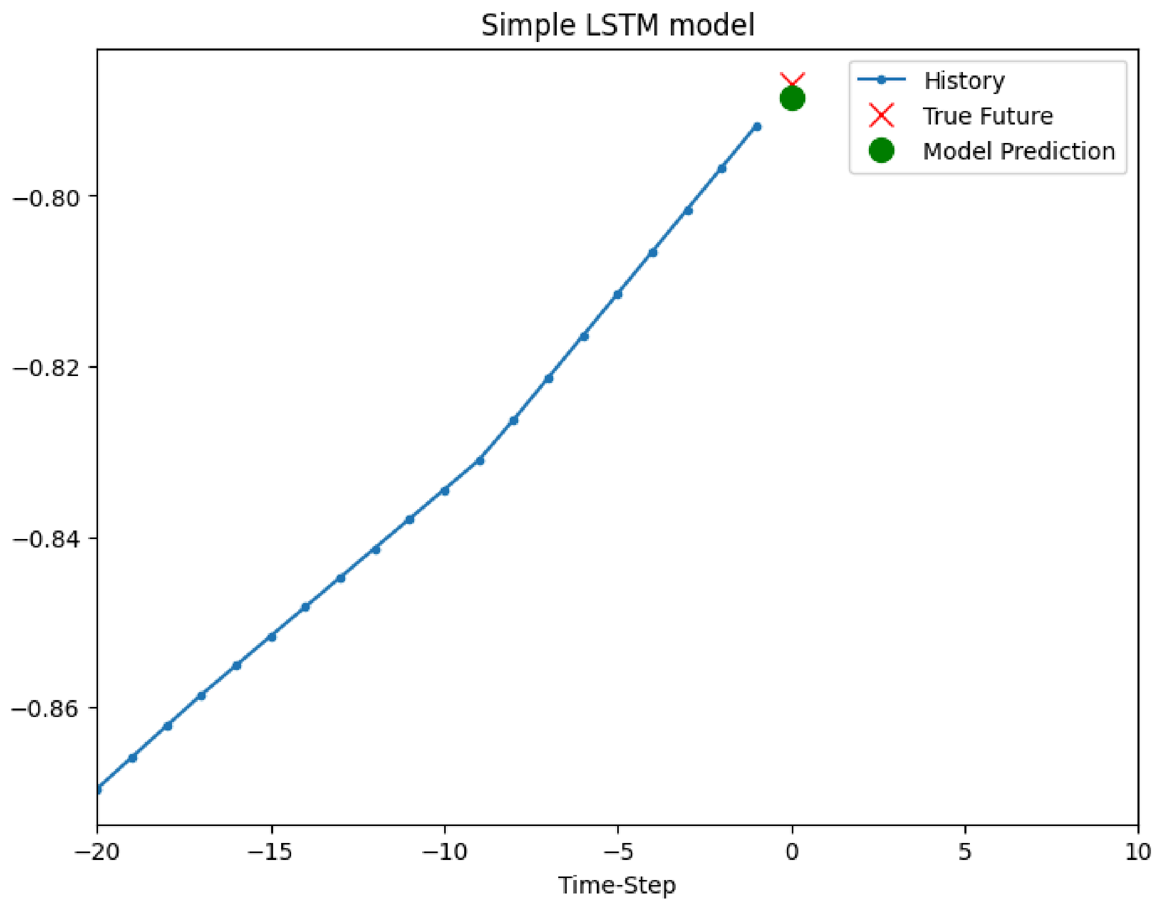
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
```




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




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

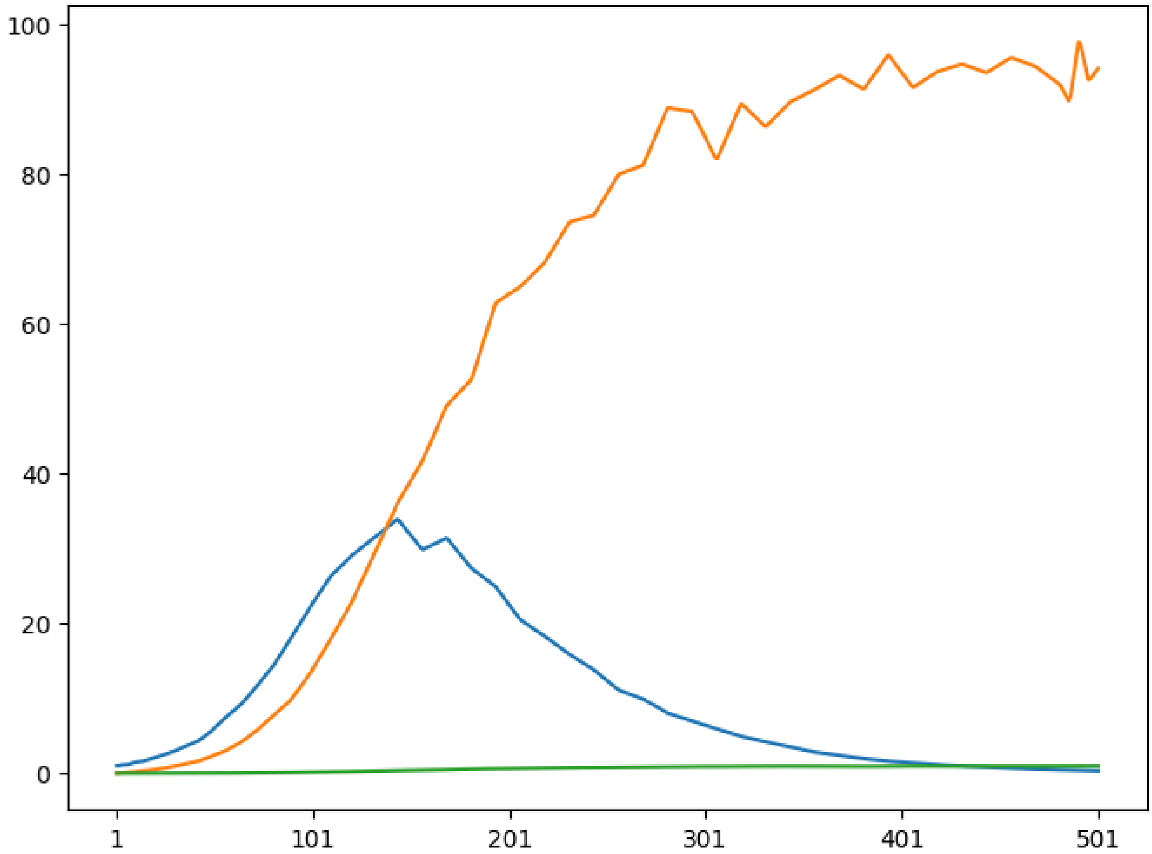


Multivariate LSTM based forecasting - Single Step

We use three variables "Infected", "Recovered", and "Deceased", to forcast "Infected" at one single day in the future.

Here a plot of the time series of the three variables for one outbreak.

In [25]:
```
dfInfected.loc[1,:].plot()
dfRecovered.loc[2,:].plot()
dfDead.loc[3,:].plot()
dfInfected = dfInfected.values
dfRecovered_arr = dfRecovered.values
dfDead_arr = dfDead.values
```



We prepare the dataset.

In [26]:
```
#as before
dfInfected_train_mean = dfInfected_arr[:TRAIN_SPLIT].mean()
dfInfected_train_std = dfInfected_arr[:TRAIN_SPLIT].std()
dfInfected_data = (dfInfected_arr-dfInfected_train_mean)/dfInfected_train_std
#for Recovered
dfRecovered_train_mean = dfRecovered_arr[:TRAIN_SPLIT].mean()
dfRecovered_train_std = dfRecovered_arr[:TRAIN_SPLIT].std()
dfRecovered_data = (dfRecovered_arr-dfRecovered_train_mean)/dfRecovered_train_st
#for Dead
dfDead_train_mean = dfDead_arr[:TRAIN_SPLIT].mean()
dfDead_train_std = dfDead_arr[:TRAIN_SPLIT].std()
dfDead_data = (dfDead_arr-dfDead_train_mean)/dfDead_train_std
```

In [27]:
```
dataset = np.array([dfInfected_data, dfRecovered_data, dfDead_data])
dataset.shape
print ('\n Multivariate data shape')
print(dataset.shape)
```

```
Multivariate data shape
(3, 150, 501)
```

In [28]:
```python
def multivariate_data(dataset, target, start_series, end_series, history_size,
                      target_size, step, single_step=False):
    data = []
    labels = []
    start_index = history_size
    end_index = len(dataset[0][0]) - target_size
    for c in range(start_series, end_series):
        for i in range(start_index, end_index):
            indices = range(i-history_size, i, step)
            one = dataset[0][c][indices]
            two = dataset[1][c][indices]
            three = dataset[2][c][indices]
            data.append(np.transpose(np.array([one, two, three])))

            if single_step:
                labels.append(target[c][i+target_size])
            else:
                labels.append(np.transpose(target[c][i:i+target_size]))
    return np.array(data), np.array(labels)
```

We get training and valdation data for time series with a `past_history = 20` days for every other day (`STEP = 2`) and want to predict the "Infected" five days ahead (`future_target = 5`).

In [29]:
```python
past_history = 20
future_target = 5
STEP = 2

x_train_single, y_train_single = multivariate_data(dataset, dfInfected_data, 0,
                                                   past_history, future_target,
                                                   single_step=True)
x_val_single, y_val_single = multivariate_data(dataset, dfInfected_data, TRAIN_S
                                               past_history, future_target, STEP
                                               single_step=True)
```

In [30]:
```python
print ('Single window of past history : {}'.format(x_train_single[0].shape))
print(dataset.shape)
```

```
Single window of past history : (10, 3)
(3, 150, 501)
```

As before, batching and resampling; the dataset is repeated indefinitely.

In [31]:
```python
train_data_single = tf.data.Dataset.from_tensor_slices((x_train_single, y_train_
train_data_single = train_data_single.cache().shuffle(BUFFER_SIZE).batch(BATCH_S

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

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

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

```
single_step_model.summary()
x_train_single.shape[-2:]
```

```
C:\Users\kemal\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.10_qbz5n2
kfra8p0\LocalCache\local-packages\Python310\site-packages\keras\src\layers\rnn\rn
n.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a lay
er. When using Sequential models, prefer using an `Input(shape)` object as the fi
rst layer in the model instead.
  super().__init__(**kwargs)
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
lstm_1 (LSTM)	(None, 32)	4,608
dense_1 (Dense)	(None, 1)	33

Total params: 4,641 (18.13 KB)

Trainable params: 4,641 (18.13 KB)

Non-trainable params: 0 (0.00 B)

Out[32]: (10, 3)

In [33]:
```
for x, y in val_data_single.take(1):
    print(single_step_model.predict(x).shape)
print ('\n Number of traing data points')
print (x_train_single.shape[0])
print ('\n Number of test data points')
print (x_val_single.shape[0])
```

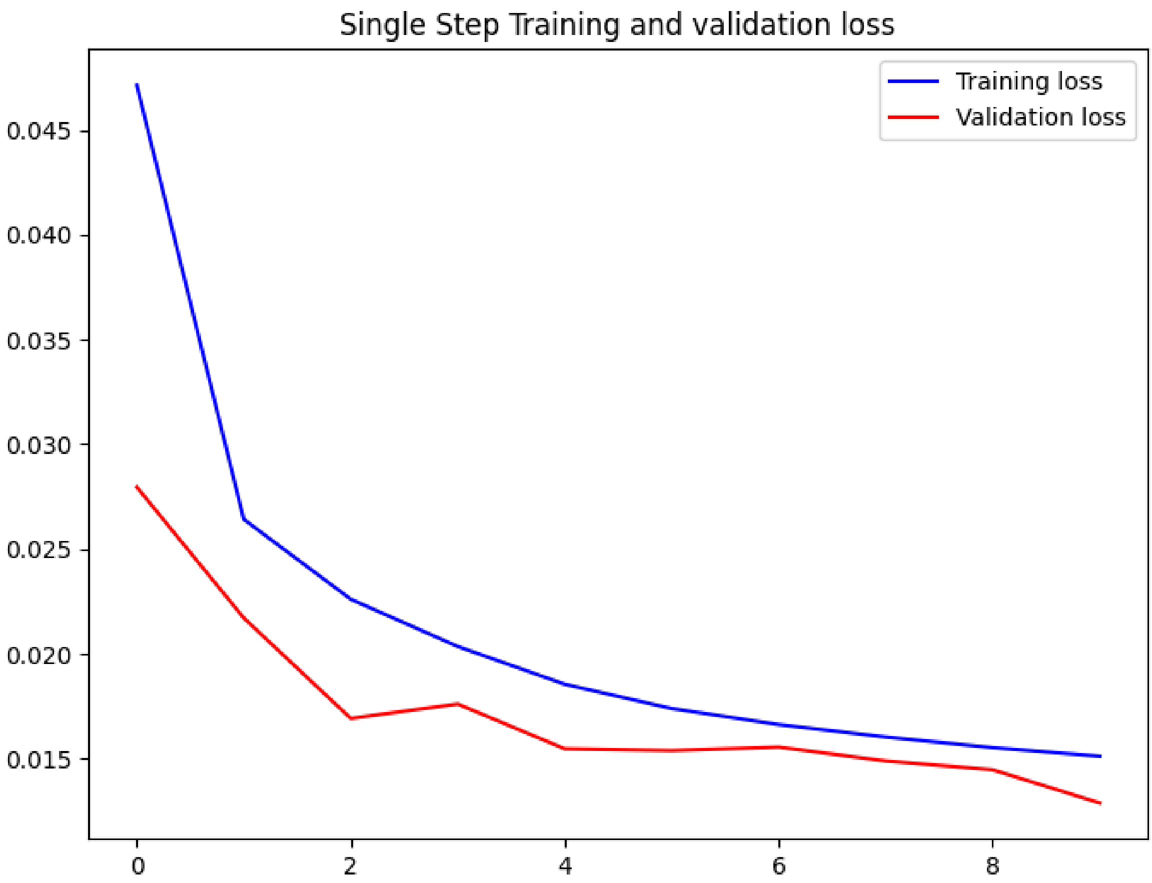
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
(256, 1)

 Number of traing data points
64260

 Number of test data points
7140
```

In [34]:
```
single_step_history = single_step_model.fit(train_data_single, epochs=EPOCHS,
                                            steps_per_epoch=EVALUATION_INTERVAL,
                                            validation_data=val_data_single,
                                            validation_steps=50)
```

```
Epoch 1/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 16s 7ms/step - loss: 0.0936 - val_loss: 0.0280
Epoch 2/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - loss: 0.0279 - val_loss: 0.0217
Epoch 3/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - loss: 0.0232 - val_loss: 0.0169
Epoch 4/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 17s 8ms/step - loss: 0.0208 - val_loss: 0.0176
Epoch 5/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - loss: 0.0189 - val_loss: 0.0155
Epoch 6/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - loss: 0.0176 - val_loss: 0.0154
Epoch 7/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - loss: 0.0168 - val_loss: 0.0155
Epoch 8/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 15s 8ms/step - loss: 0.0161 - val_loss: 0.0149
Epoch 9/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - loss: 0.0156 - val_loss: 0.0145
Epoch 10/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - loss: 0.0152 - val_loss: 0.0129
```

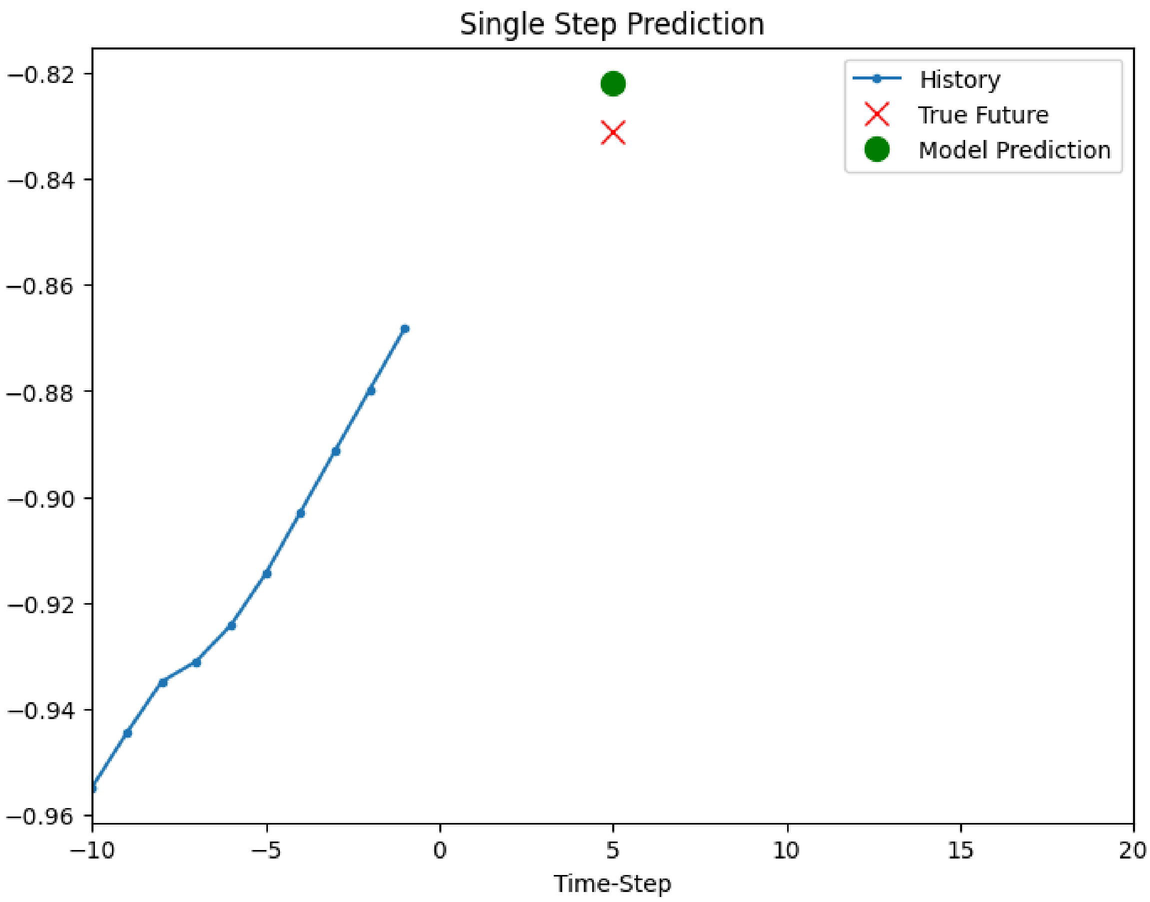
In [35]:
```python
def plot_train_history(history, title):
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs = range(len(loss))
    plt.figure()
    plt.plot(epochs, loss, 'b', label='Training loss')
    plt.plot(epochs, val_loss, 'r', label='Validation loss')
    plt.title(title)
    plt.legend()
    plt.show()
```

In [36]:
```python
plot_train_history(single_step_history,'Single Step Training and validation loss
```

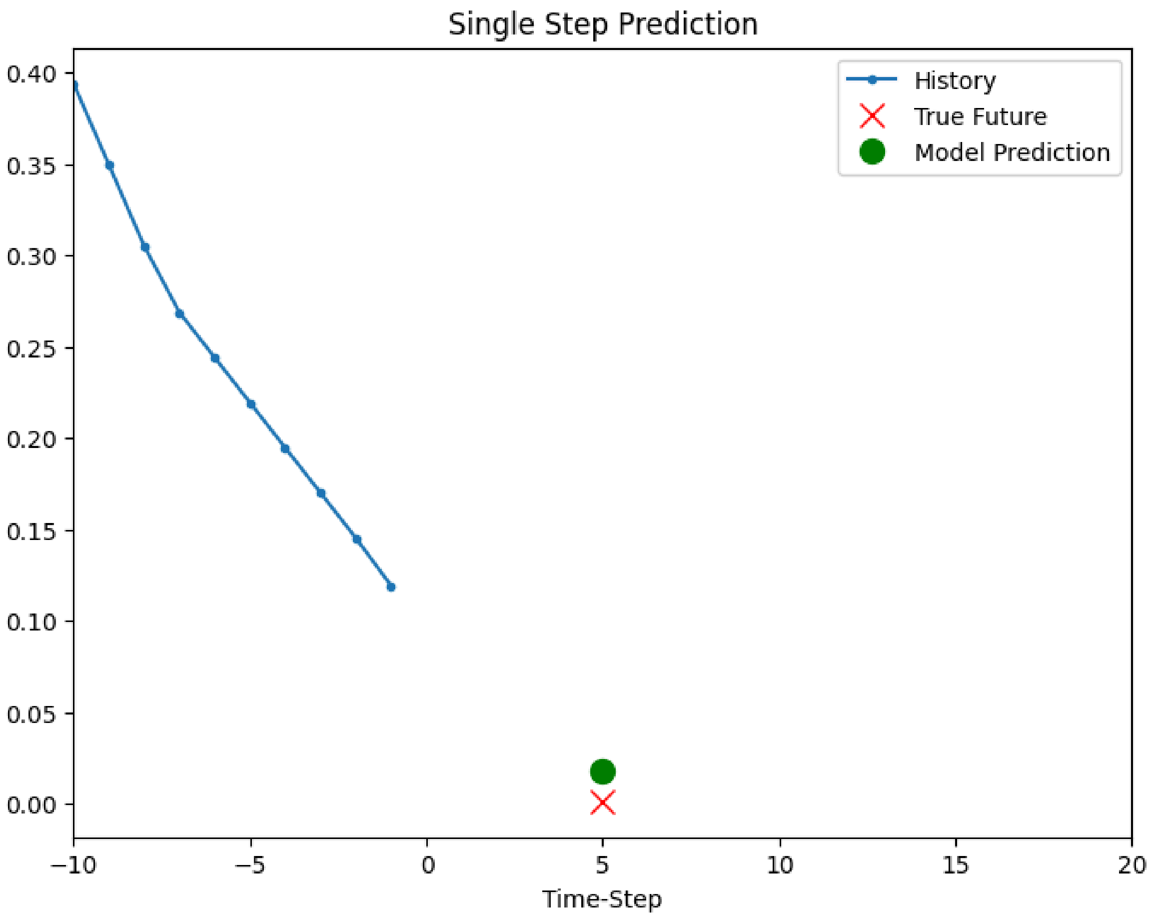


In [37]:
```
for x, y in val_data_single.take(3):
    plot = show_plot([x[0][:, 0].numpy(), y[0].numpy(),
                    single_step_model.predict(x)[0]], future_target,
                   'Single Step Prediction')
    plot.show()
```

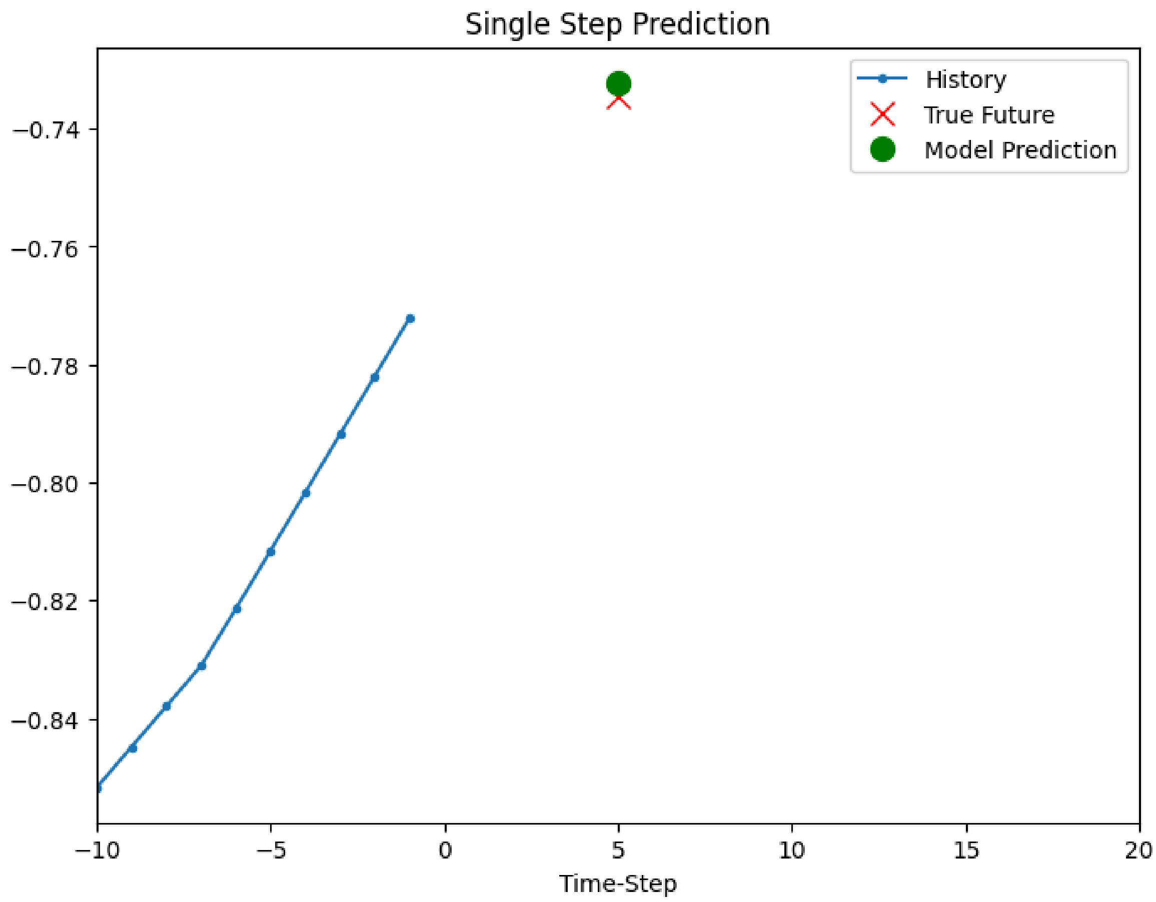
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
```



8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step



8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step



Multivariate LSTM - Multiple Steps

Still, we use a series of observed values of the three variables "Infected", "Recovered", and "Deceased" (`past_history = 40, STEP =2`), but now to forcast the "Infected" values for a series day in the future (`future_target = 10`).

In [38]:
```
past_history = 40
future_target = 10
STEP =2
x_train_multi, y_train_multi = multivariate_data(dataset, dfInfected_data, 0, TR
                                                 past_history, future_target,
x_val_multi, y_val_multi = multivariate_data(dataset, dfInfected_data, TRAIN_SPL
                                             past_history, future_target, STE
```

In [39]:
```
print ('Single window of past history : {}'.format(x_train_multi[0].shape))
print ('\nTarget window to predict : {}'.format(y_train_multi[0].shape))
print ('\nNumber of traing data points: {}'.format(x_train_multi.shape[0]))
print ('\nNumber of test data points: {}'.format(x_val_multi.shape[0]))
```

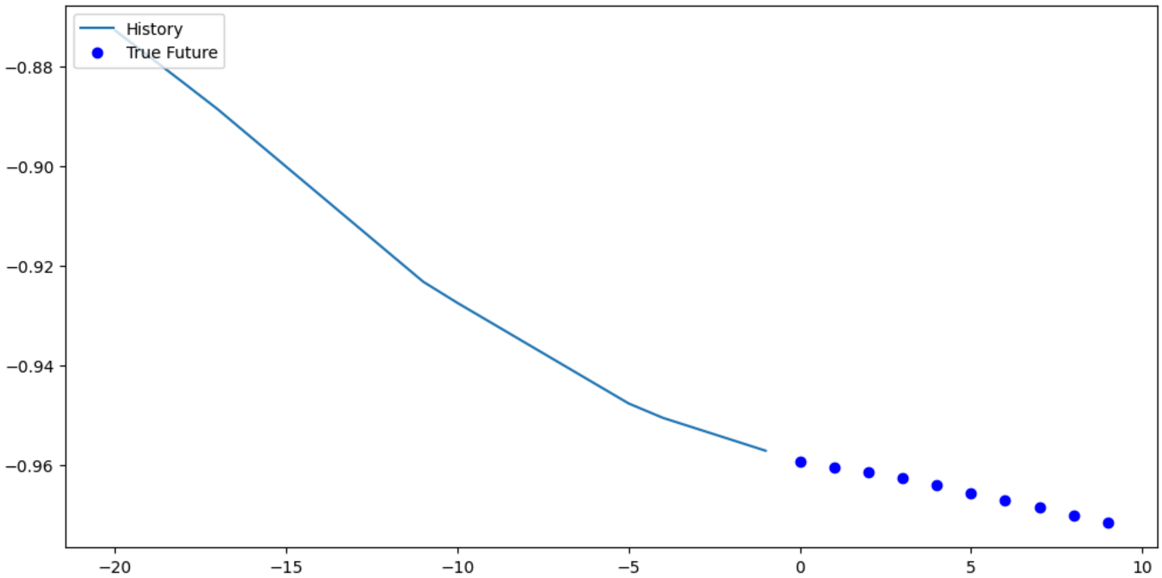
```
Single window of past history : (20, 3)

Target window to predict : (10,)

Number of traing data points: 60885

Number of test data points: 6765
```

As before, batching and resampling; the dataset is repeated indefinitely.

In [40]:
```
train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_multi, y_train_mu
train_data_multi = train_data_multi.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZ

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

In [41]:
```
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[:, 0]), label='History')
    plt.plot(np.arange(num_out), np.array(true_future), 'bo', label='True Future
    if prediction.any():
        plt.plot(np.arange(num_out), np.array(prediction), 'ro', label='Predicte
    plt.legend(loc='upper left')
    plt.show()
```

In [42]:
```
for x, y in train_data_multi.take(1):
    multi_step_plot(x[0], y[0], np.array([0]))
```



Now we bild a model with two LSTM layers.

In [43]:
```
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(future_target))

multi_step_model.compile(optimizer=tf.keras.optimizers.RMSprop(clipvalue=1.0), l
multi_step_model.summary()
x_train_multi.shape[-2:]
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
lstm_2 (LSTM)	(None, 20, 32)	4,608
lstm_3 (LSTM)	(None, 16)	3,136
dense_2 (Dense)	(None, 10)	170

Total params: 7,914 (30.91 KB)

Trainable params: 7,914 (30.91 KB)

Non-trainable params: 0 (0.00 B)

Out[43]: (20, 3)

In [44]:
```
for x, y in val_data_multi.take(1):
    print (multi_step_model.predict(x).shape)
```

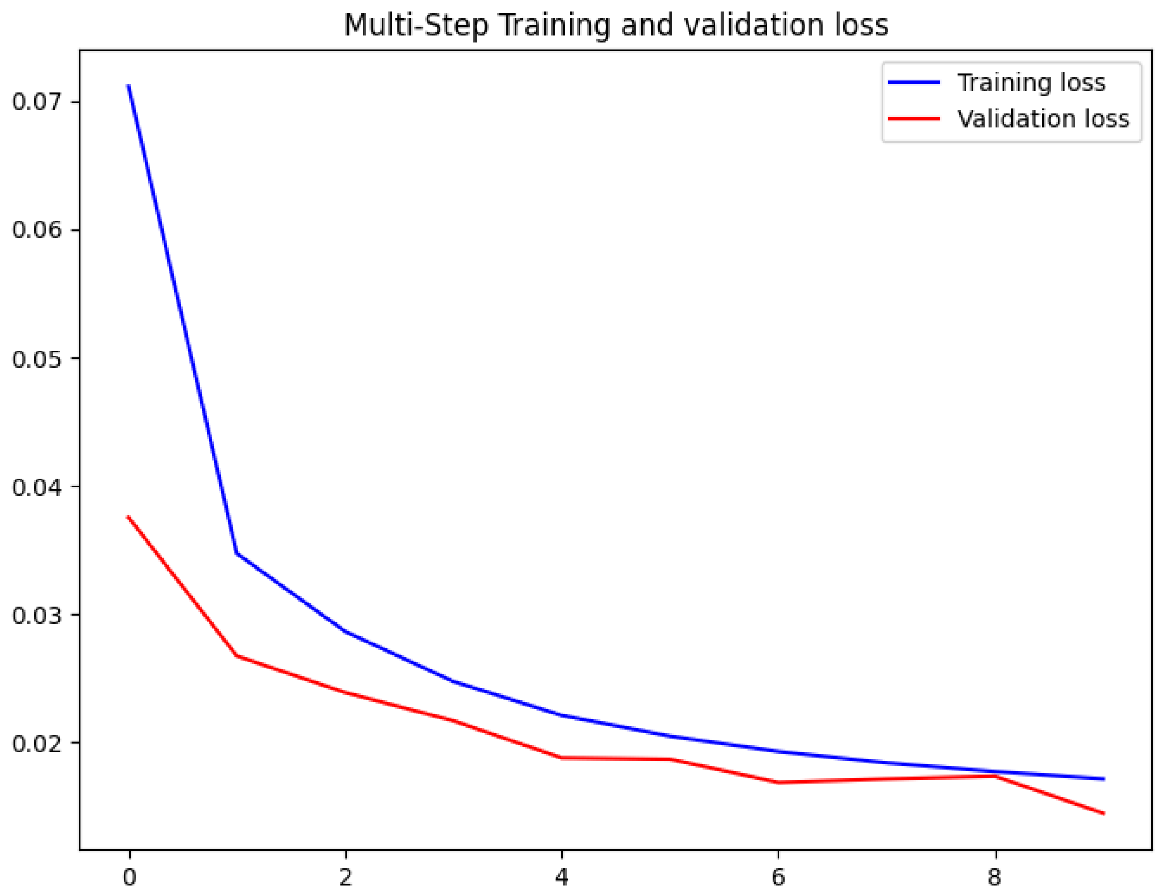
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
(256, 10)
```

The training time is longer for this more complex model.

In [45]:
```
multi_step_history = multi_step_model.fit(train_data_multi, epochs=EPOCHS,
                                          steps_per_epoch=EVALUATION_INTERVAL,
                                          validation_data=val_data_multi,
                                          validation_steps=50)
```

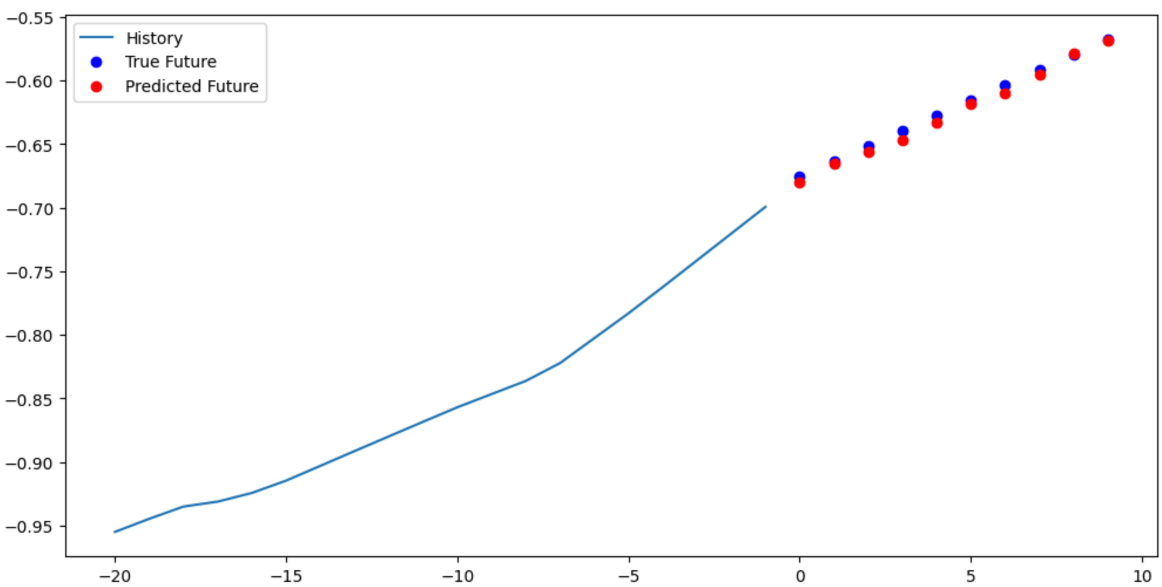
```
Epoch 1/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 37s 17ms/step - loss: 0.1355 - val_loss: 0.0376
Epoch 2/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 33s 17ms/step - loss: 0.0369 - val_loss: 0.0268
Epoch 3/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 36s 18ms/step - loss: 0.0297 - val_loss: 0.0239
Epoch 4/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 35s 18ms/step - loss: 0.0255 - val_loss: 0.0217
Epoch 5/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 35s 18ms/step - loss: 0.0227 - val_loss: 0.0188
Epoch 6/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 35s 18ms/step - loss: 0.0209 - val_loss: 0.0187
Epoch 7/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 35s 18ms/step - loss: 0.0195 - val_loss: 0.0169
Epoch 8/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 35s 18ms/step - loss: 0.0187 - val_loss: 0.0172
Epoch 9/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 35s 18ms/step - loss: 0.0179 - val_loss: 0.0174
Epoch 10/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 35s 18ms/step - loss: 0.0173 - val_loss: 0.0145
```

In [46]:
```
plot_train_history(multi_step_history, 'Multi-Step Training and validation loss'
```

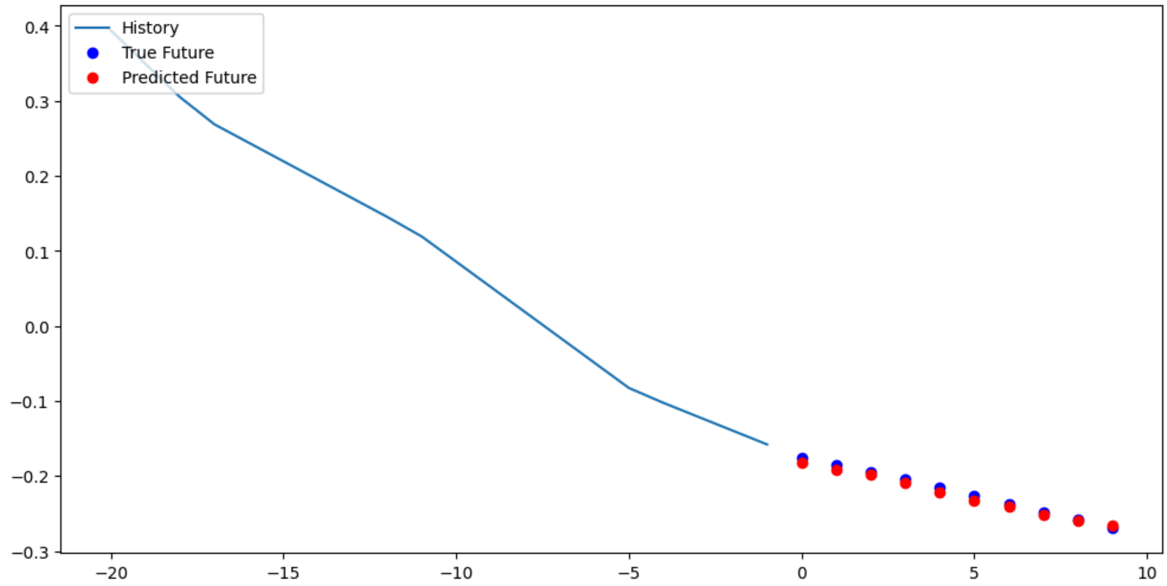


In [47]:
```
for x, y in val_data_multi.take(3):
    multi_step_plot(x[0], y[0], multi_step_model.predict(x)[0])
```

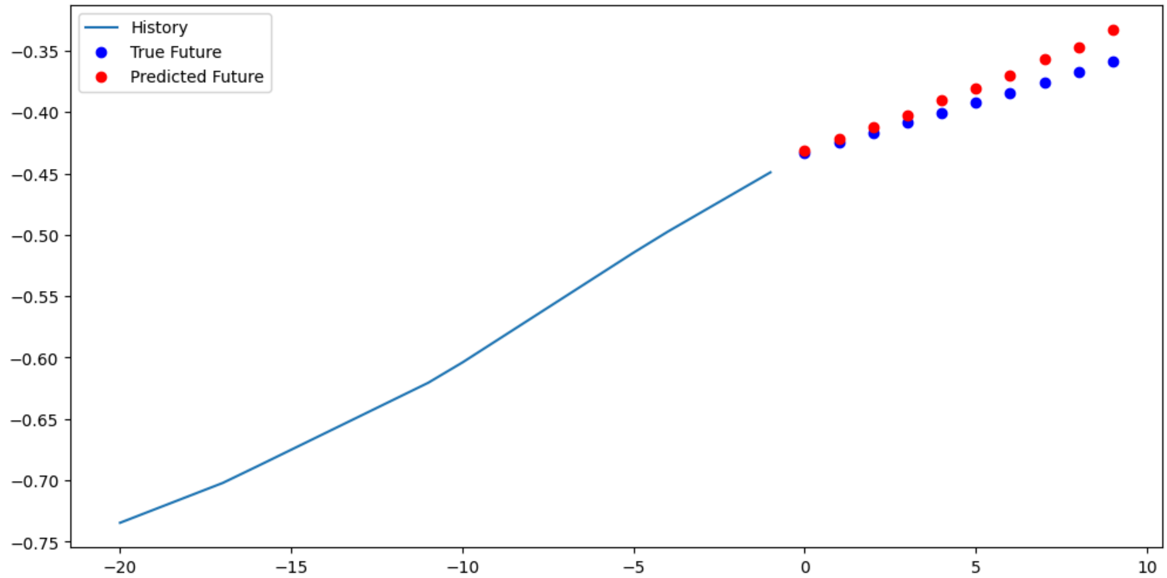
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step
```




```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
```



8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step



MODIEFIED MODELS

In this chapter, i have made modifications to the code featured in the previous chapters. What i did is that i copied over chapters 4, 5 and 6 and collapsed them together so that they would have their own notebook cell in this chapter. This was done to make the changes easier to highlight and understand even though the notebook might be less "modular".

Below, you will find every change that i made. A lot of them were already suggested in the Moodle page but some of them were also made solely by me as i found some small opportunities to change the model and the outcome of the model.

A lot of the conclusions that i make from my experiments are visual. This means that i try to compare the accuracy from my graphs and logs to the already provided models and try to draw conclusions from that instead of tracking some kind of metric as that might not tell us the full story of how the model performs over epochs. the logs will instead give me a clearer picture.

- SWAPPING FROM LSTM TO GRU

  The first thing i did was to try and change the layers from LSTM to GRU. This was very simple as i could just change which `tf` function we call when defining the `simple_GRU_model`. We change the initial layer from `tf.keras.layers.LSTM` to `tf.keras.layers.GRU`. By looking at the logs from TensorFlow we can see that the amount of decreased from 320 to 264 for the simple model But i could not see clear difference in how the model performed in the logs. However, i felt that because we have quite a significant lesser amount of parameters with arguably the same performance, i sticked with GRU layer instead of LSTM as this gives us improvements on model complexity and training time.

- ADDING MORE DENSE LAYERS

  The second thing i did was to try and add ReLu-activated dense layers after the initial GRU layer. My assumption was that additional non-linear transformations could help the model capture more complex temporal patterns in the data. This was almost as easy as the previous step. The only thing i had to do was to add `tf.keras.layers.Dense(16, activation='relu'),` right below where i defined my GRU layer. I experimented with different activation functions and different amounts of units. What i in the end found was that ReLu worked quite well and by doing visual inspection i could see that having two layers with 16 and 8 units respectively gave a bit better accuracy as we were making better predictions on the true future point. Even though adding more dense layers increased the amount of parameters, as seen by the TensorFlow logs, this is in my opinion a worthy trade-off because we could still consider this model "lightweight" as it can still be trained on my consumer based laptop on CPU while getting quite a nice accuracy and performance boost overall from it.

- INCREASING HISTORY LENGTH

  I also tried increasing the history length. This was probably the easiest modification to make (not that the other ones were especially hard) because i only needed to change the `univariate_past_history` variable that was defined higher up in the code. I pasted that code over to this section of the notebook as well so that i have easier control over it. By default, it was set to 20 but i could change this to any valid int-value. First, i wanted to make a visual validation that this would make a difference so i first decreased it to 3 and got very bad results but the training time went from 2m30s to 45s for the simple model. This kind of behavior was expected so i decided to increase the variable. First i tried just doubling it from the default so i defined it as 40 days and the model convergence was much slower but we got much better results in the accuracy. This far, this is the factor that has made the biggest difference in performance which makes sence because increasing the history would give the model more context to make better future predictions.

Univariate GRU based forecasting

In [48]:
```
univariate_past_history = 2 #days
univariate_future_target = 0 #current day

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, len(uni_data),
                                       univariate_past_history,
                                       univariate_future_target)


print (x_train_uni.shape)
print (y_train_uni.shape)
x_train_uni.dtype

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_SIZ

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

train_univariate


# defining the model using RU instead of LSTM
simple_GRU_model = tf.keras.models.Sequential([
    tf.keras.layers.GRU(8, input_shape=x_train_uni.shape[-2:]),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(1)
])

simple_GRU_model.compile(optimizer='adam', loss='mae')
simple_GRU_model.summary()
x_train_uni.shape[-2:]

for x, y in val_univariate.take(1):
    print(simple_GRU_model.predict(x).shape)
    print(y.shape)


EVALUATION_INTERVAL = 2000
EPOCHS = 10

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

for x, y in val_univariate.take(3):
    plot = show_plot([x[0].numpy(), y[0].numpy(), simple_GRU_model.predict(x)[0]
    plot.show()
```

```python
# "Show diagrams with predictions around the peak of infection"

# Flatten
flat_data = uni_data[0]

# Get top 3 peak indices (excluding the very start to allow for history)
peak_indices = np.argpartition(flat_data[univariate_past_history:], -3)[-3:] + u
peak_indices = sorted(peak_indices)

for i, peak_idx in enumerate(peak_indices):
    start_idx = peak_idx - univariate_past_history
    input_seq = flat_data[start_idx:peak_idx].reshape(1, -1, 1)

    # Predict the value at the peak using your model
    predicted = simple_GRU_model.predict(input_seq)[0]
    true_value = flat_data[peak_idx]

    # Use your existing show_plot function
    plot = show_plot([input_seq[0], true_value, predicted], 0, f'Prediction arou
    plot.show()
```

```
(67365, 2, 1)
(67365,)
```

Model: "sequential_3"

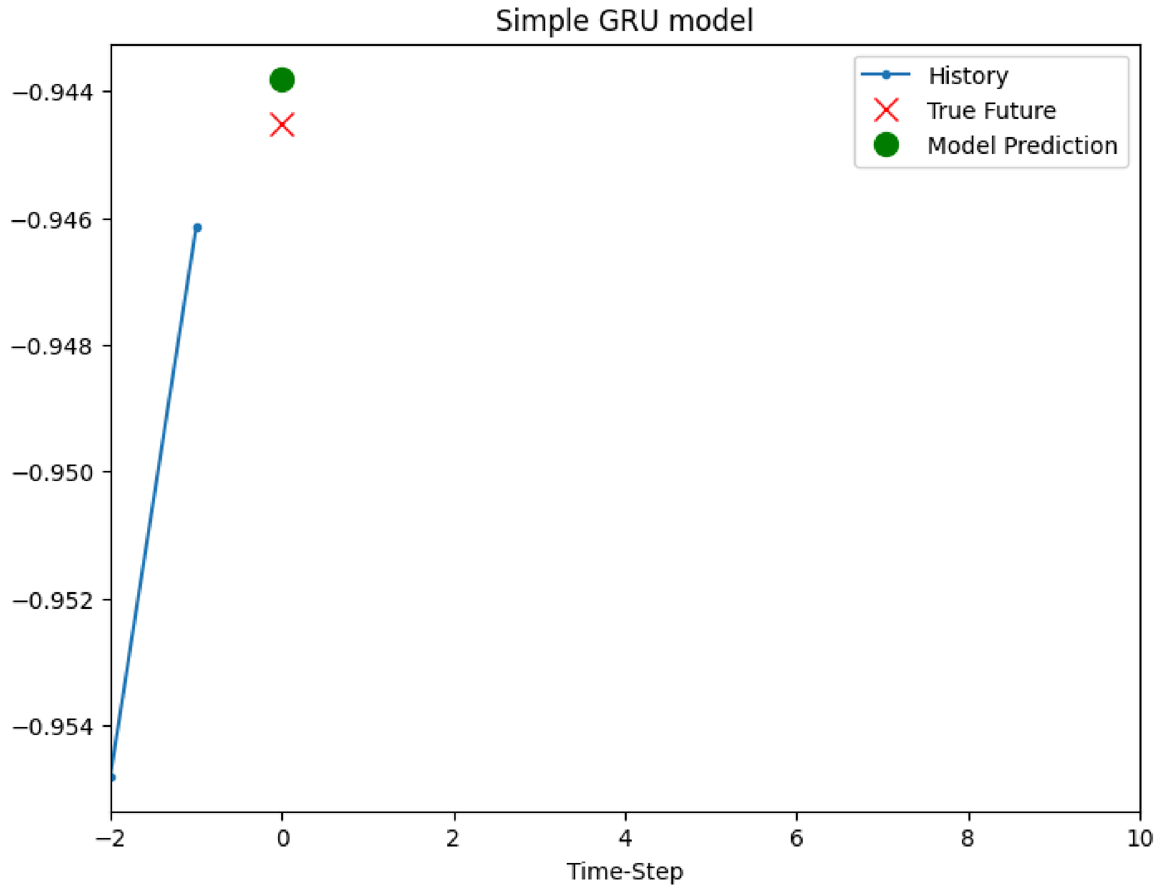
Layer (type)	Output Shape	Param #
gru (GRU)	(None, 8)	264
dense_3 (Dense)	(None, 16)	144
dense_4 (Dense)	(None, 8)	136
dense_5 (Dense)	(None, 1)	9

Total params: 553 (2.16 KB)

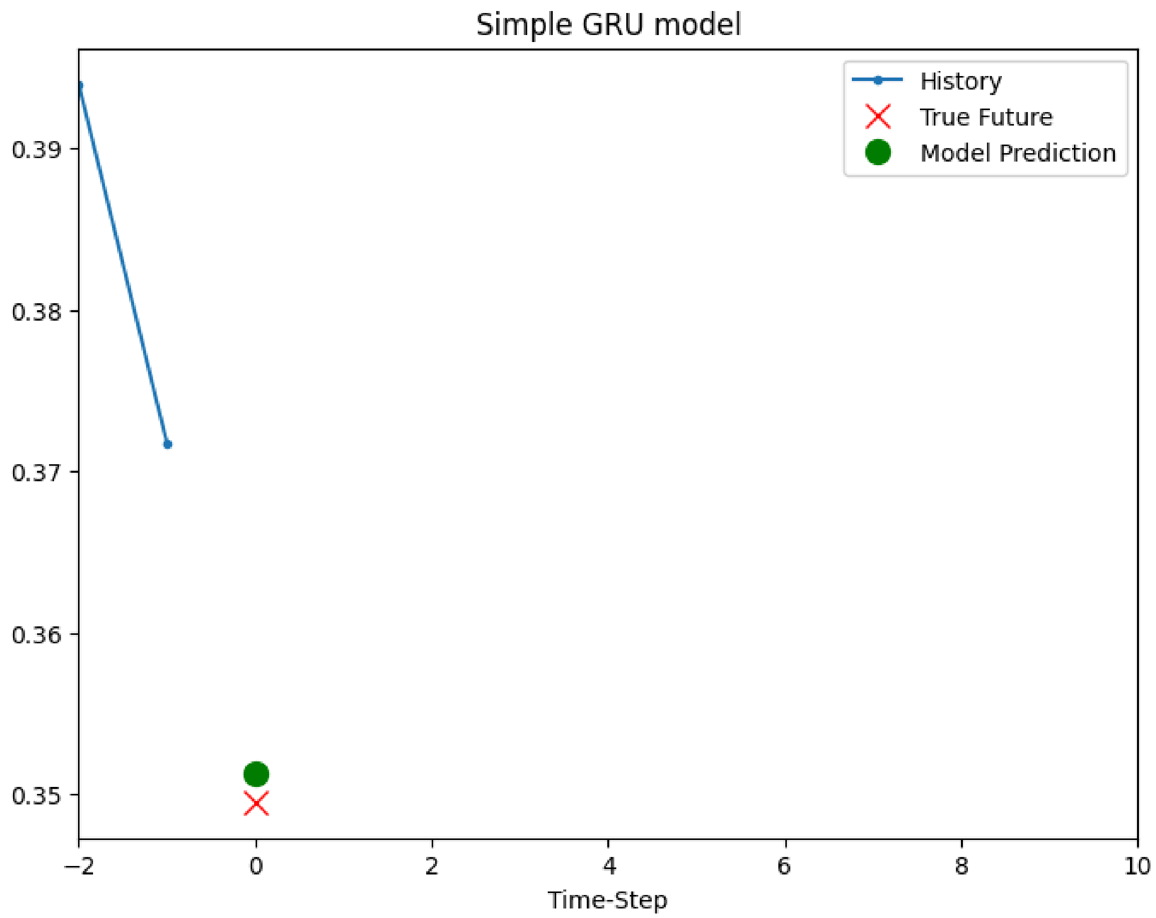
Trainable params: 553 (2.16 KB)

Non-trainable params: 0 (0.00 B)

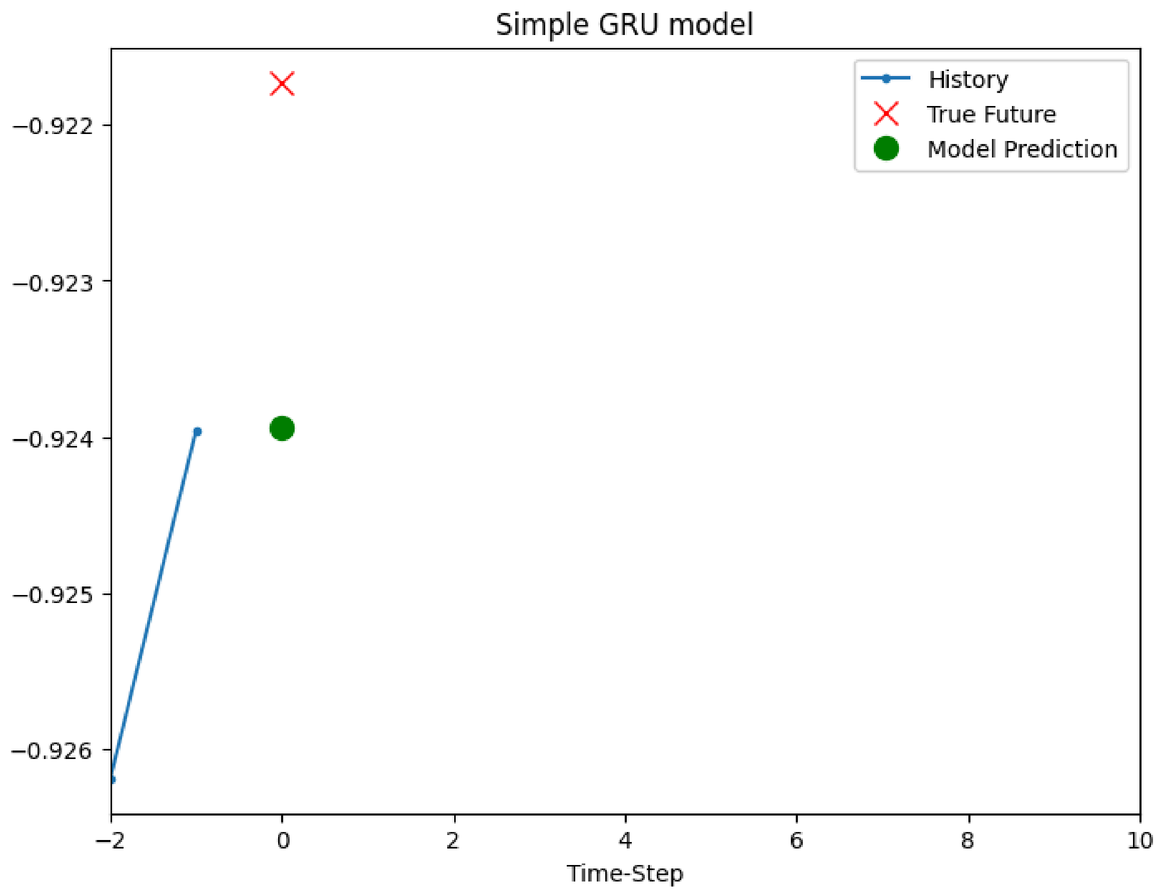
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
(256, 1)
(256,)
Epoch 1/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 8s 3ms/step - loss: 0.1854 - val_loss: 0.0110
Epoch 2/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0104 - val_loss: 0.0059
Epoch 3/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0052 - val_loss: 0.0049
Epoch 4/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0041 - val_loss: 0.0025
Epoch 5/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0035 - val_loss: 0.0025
Epoch 6/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0034 - val_loss: 0.0026
Epoch 7/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0032 - val_loss: 0.0035
Epoch 8/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0029 - val_loss: 0.0023
Epoch 9/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0030 - val_loss: 0.0023
Epoch 10/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0030 - val_loss: 0.0016
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
```




```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
```

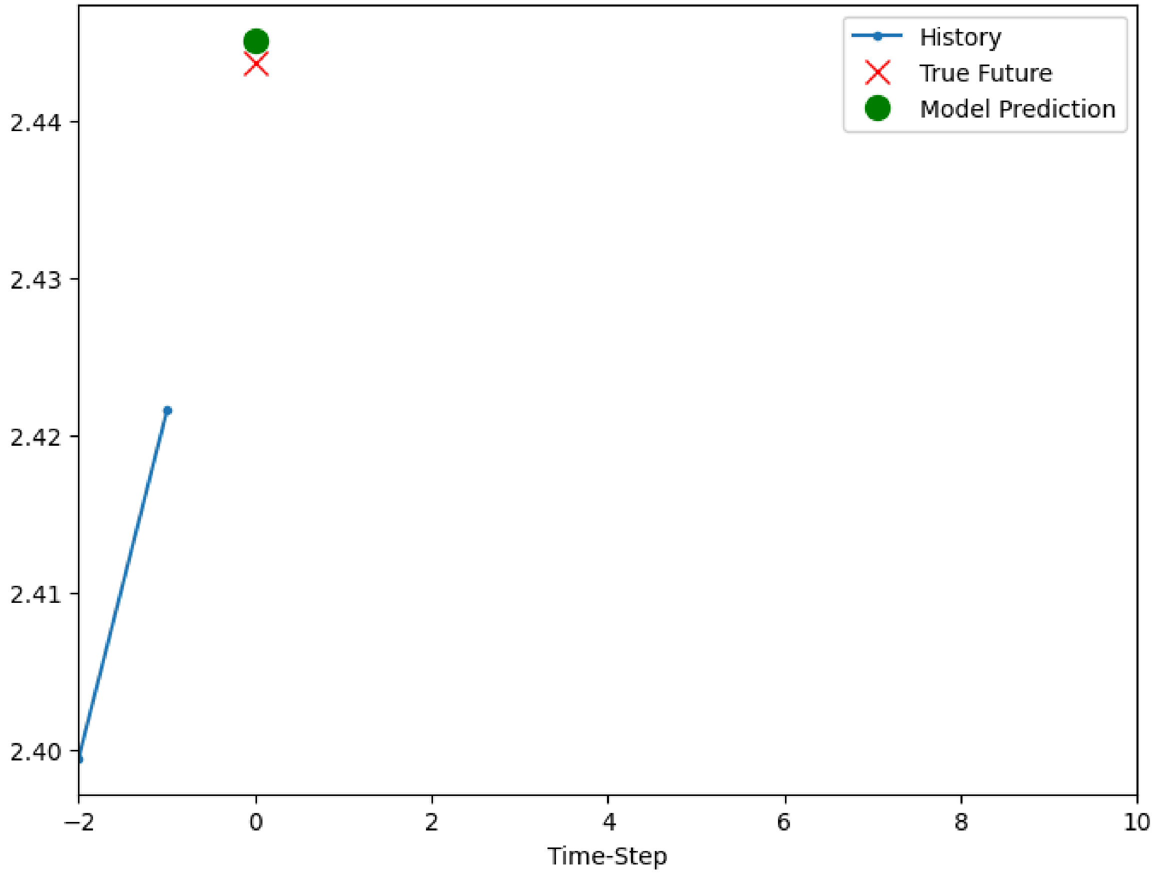


`8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step`



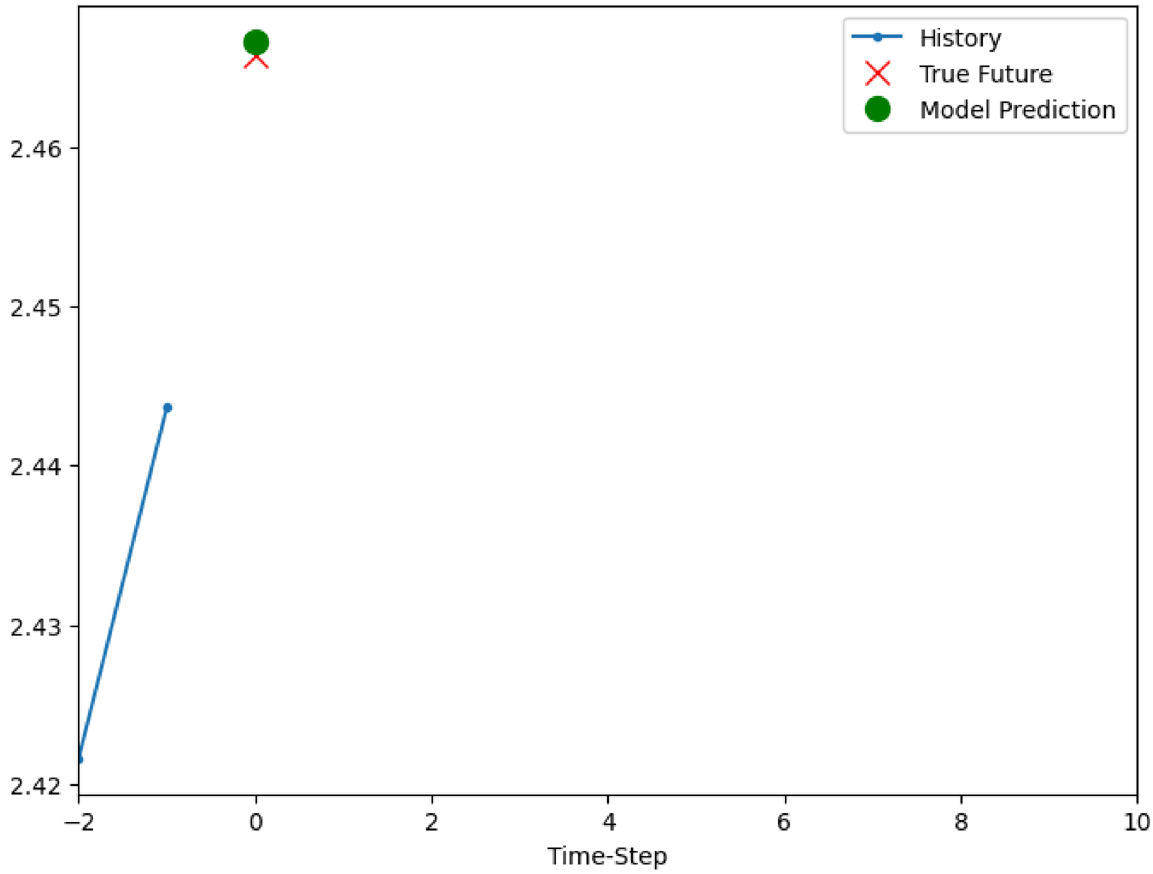
`1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 243ms/step`

Prediction around infection peak #1

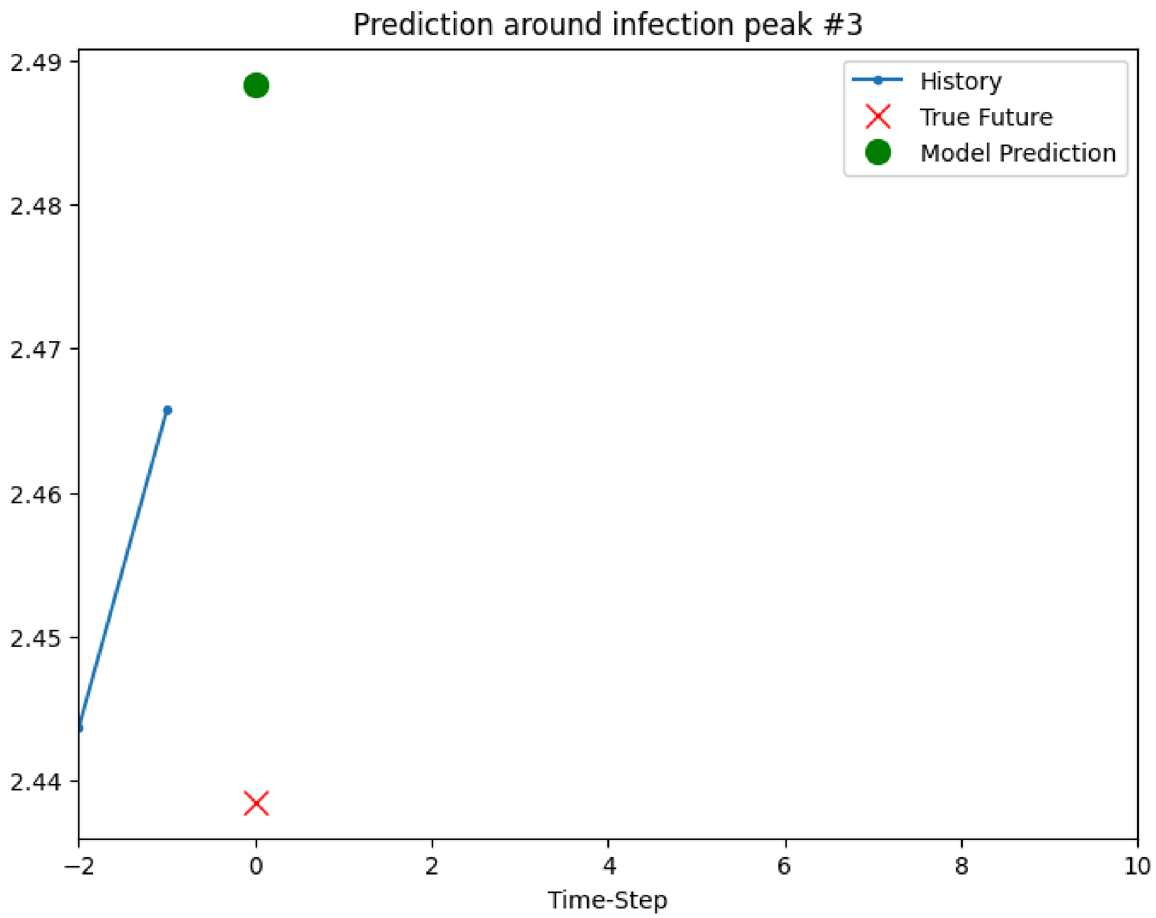



```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step
```

Prediction around infection peak #2




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step
```



Multivariate GRU based forecasting - Single Step

In [49]:
```
#as before
dfInfected_train_mean = dfInfected_arr[:TRAIN_SPLIT].mean()
dfInfected_train_std = dfInfected_arr[:TRAIN_SPLIT].std()
dfInfected_data = (dfInfected_arr-dfInfected_train_mean)/dfInfected_train_std
#for Recovered
dfRecovered_train_mean = dfRecovered_arr[:TRAIN_SPLIT].mean()
dfRecovered_train_std = dfRecovered_arr[:TRAIN_SPLIT].std()
dfRecovered_data = (dfRecovered_arr-dfRecovered_train_mean)/dfRecovered_train_st
#for Dead
dfDead_train_mean = dfDead_arr[:TRAIN_SPLIT].mean()
dfDead_train_std = dfDead_arr[:TRAIN_SPLIT].std()
dfDead_data = (dfDead_arr-dfDead_train_mean)/dfDead_train_std

dataset = np.array([dfInfected_data, dfRecovered_data, dfDead_data])
dataset.shape
print ('\n Multivariate data shape')
print(dataset.shape)

def multivariate_data(dataset, target, start_series, end_series, history_size,
                      target_size, step, single_step=False):
    data = []
    labels = []
    start_index = history_size
    end_index = len(dataset[0][0]) - target_size
    for c in range(start_series, end_series):
        for i in range(start_index, end_index):
            indices = range(i-history_size, i, step)
            one = dataset[0][c][indices]
```

```
            two = dataset[1][c][indices]
            three = dataset[2][c][indices]
            data.append(np.transpose(np.array([one, two, three])))

            if single_step:
                labels.append(target[c][i+target_size])
            else:
                labels.append(np.transpose(target[c][i:i+target_size]))
    return np.array(data), np.array(labels)

past_history = 2
future_target = 5
STEP = 2

x_train_single, y_train_single = multivariate_data(dataset, dfInfected_data, 0,
                                                   past_history, future_target,
                                                   single_step=True)
x_val_single, y_val_single = multivariate_data(dataset, dfInfected_data, TRAIN_S
                                               past_history, future_target, STEP
                                               single_step=True)

train_data_single = tf.data.Dataset.from_tensor_slices((x_train_single, y_train_
train_data_single = train_data_single.cache().shuffle(BUFFER_SIZE).batch(BATCH_S

val_data_single = tf.data.Dataset.from_tensor_slices((x_val_single, y_val_single
val_data_single = val_data_single.batch(BATCH_SIZE).repeat()

single_step_model_GRU = tf.keras.models.Sequential()
single_step_model_GRU.add(tf.keras.layers.GRU(32, input_shape=x_train_single.sha
single_step_model_GRU.add(tf.keras.layers.Dense(1))

single_step_model_GRU.compile(optimizer=tf.keras.optimizers.RMSprop(), loss='mae
single_step_model_GRU.summary()
x_train_single.shape[-2:]

for x, y in val_data_single.take(1):
    print(single_step_model_GRU.predict(x).shape)
print ('\n Number of traing data points')
print (x_train_single.shape[0])
print ('\n Number of test data points')
print (x_val_single.shape[0])


single_step_history_GRU = single_step_model_GRU.fit(train_data_single, epochs=EP
                                            steps_per_epoch=EVALUATION_INTERVAL,
                                            validation_data=val_data_single,
                                            validation_steps=50)

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()

plot_train_history(single_step_history_GRU,'Single Step Training and validation
```

```
for x, y in val_data_single.take(3):
    plot = show_plot([x[0][:, 0].numpy(), y[0].numpy(),
                    single_step_model_GRU.predict(x)[0]], future_target,
                   'Single Step Prediction')
    plot.show()

# “Show diagrams with predictions around the peak of infection”

# Use standardized infection data for peak detection
flat_data = dfInfected_data[0] if len(dfInfected_data.shape) == 2 else dfInfecte

# Identify top 3 peaks (exclude early points to leave room for history)
peak_indices = np.argpartition(flat_data[past_history:], -3)[-3:] + past_history
peak_indices = sorted(peak_indices)

for i, peak_idx in enumerate(peak_indices):
    # Get indices with correct step size
    input_indices = range(peak_idx - past_history, peak_idx, STEP)
    one = dfInfected_data[0][input_indices]
    two = dfRecovered_data[0][input_indices]
    three = dfDead_data[0][input_indices]

    input_seq = np.transpose(np.array([one, two, three])).reshape(1, -1, 3)

    # Predict future value using multivariate GRU model
    predicted = single_step_model_GRU.predict(input_seq)[0]
    true_value = dfInfected_data[0][peak_idx]

    print(f"Peak #{i+1} — Index: {peak_idx}, True: {true_value:.3f}, Predicted:

    # Use your custom plot function (adapted for multivariate input)
    plot = show_plot([input_seq[0][:, 0], true_value, predicted], future_target,
                     f'Prediction near infection peak #{i+1}')
    plot.show()
```

```
 Multivariate data shape
(3, 150, 501)
```

Model: "sequential_4"

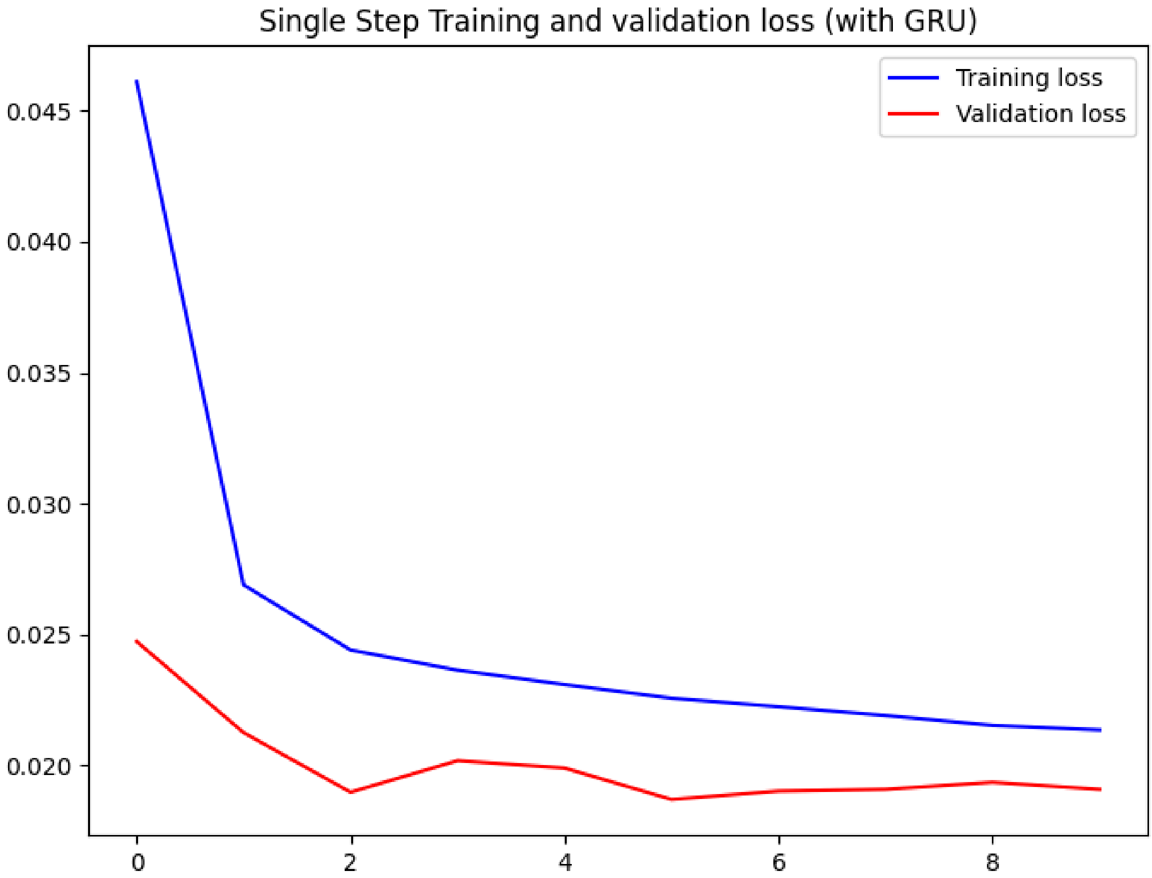
Layer (type)	Output Shape	Param #
gru_1 (GRU)	(None, 32)	3,552
dense_6 (Dense)	(None, 1)	33

Total params: 3,585 (14.00 KB)

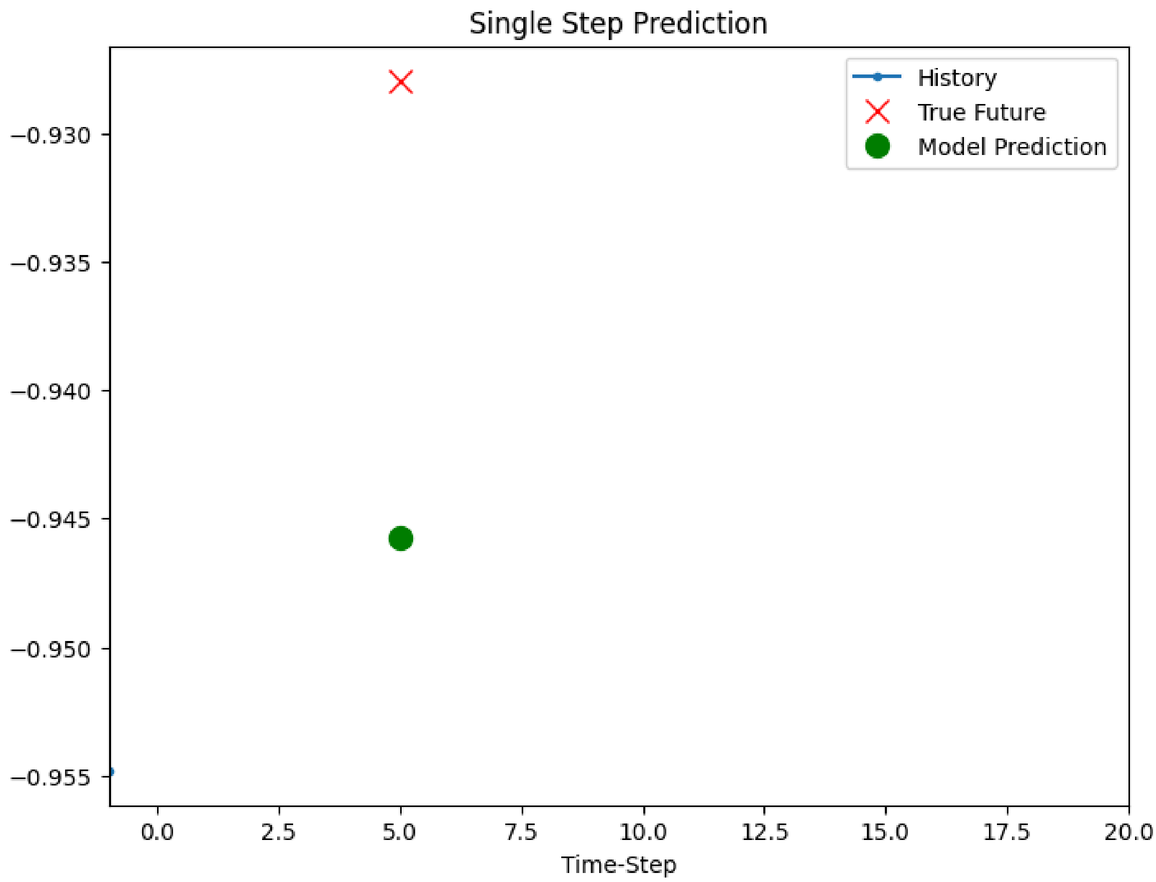
Trainable params: 3,585 (14.00 KB)

Non-trainable params: 0 (0.00 B)

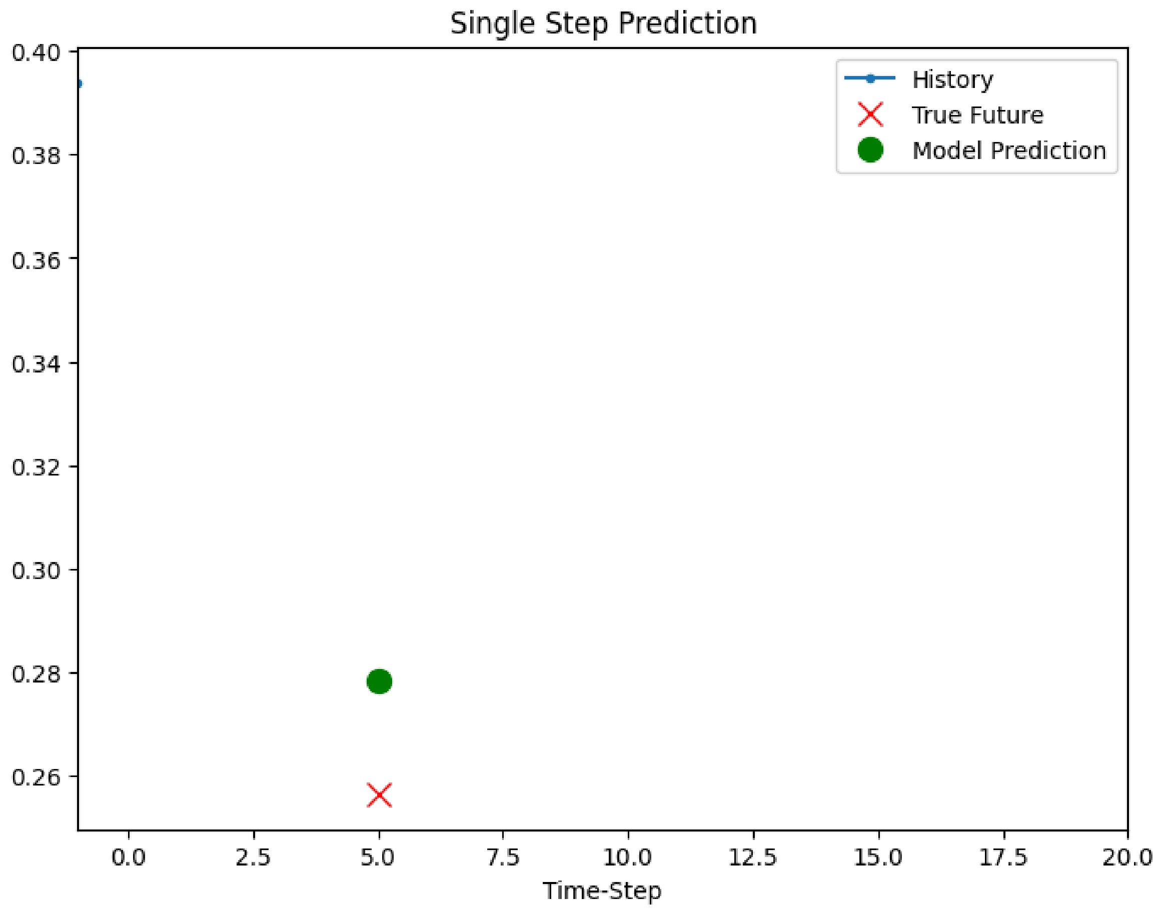
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
(256, 1)

 Number of traing data points
66690

 Number of test data points
7410
Epoch 1/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - loss: 0.0927 - val_loss: 0.0247
Epoch 2/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 0.0281 - val_loss: 0.0213
Epoch 3/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 0.0247 - val_loss: 0.0190
Epoch 4/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 0.0240 - val_loss: 0.0202
Epoch 5/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 0.0232 - val_loss: 0.0199
Epoch 6/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 0.0227 - val_loss: 0.0187
Epoch 7/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 0.0225 - val_loss: 0.0190
Epoch 8/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 0.0219 - val_loss: 0.0191
Epoch 9/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - loss: 0.0216 - val_loss: 0.0194
Epoch 10/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - loss: 0.0215 - val_loss: 0.0191
```



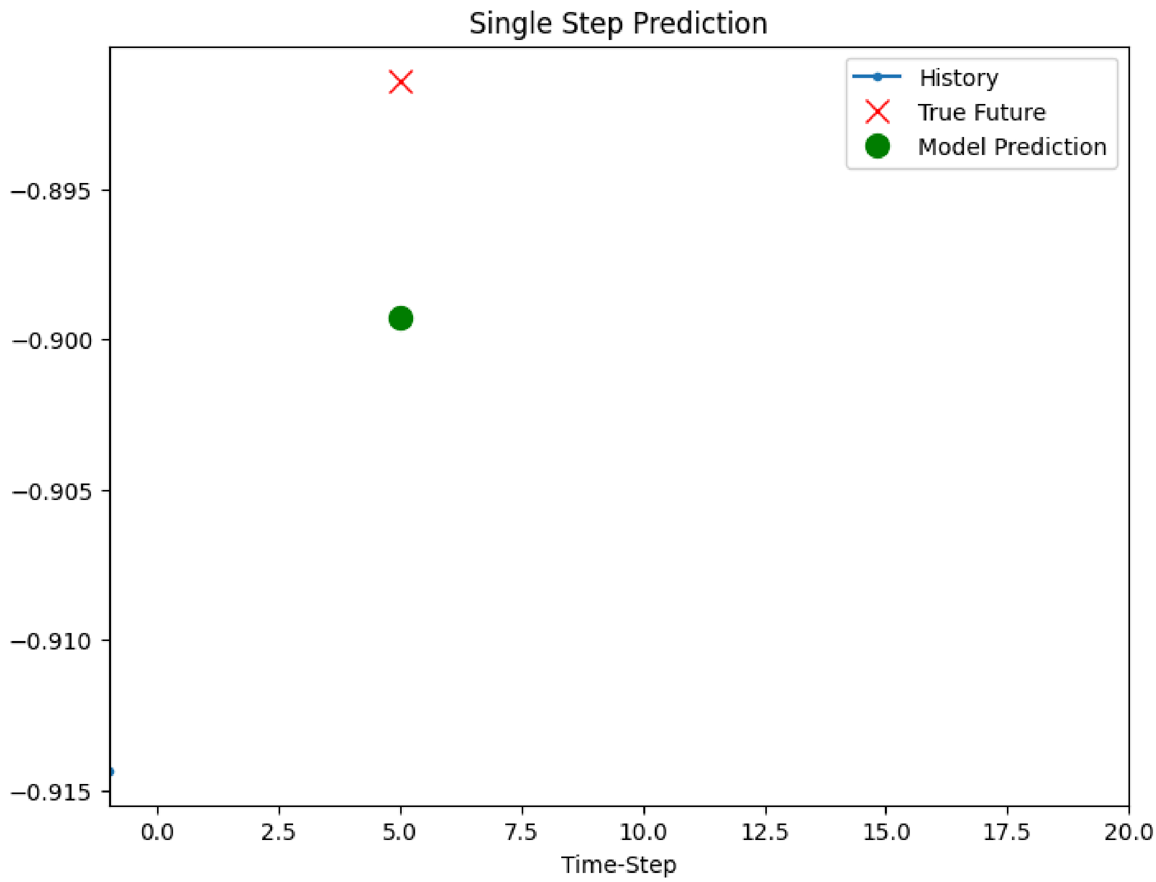

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
```



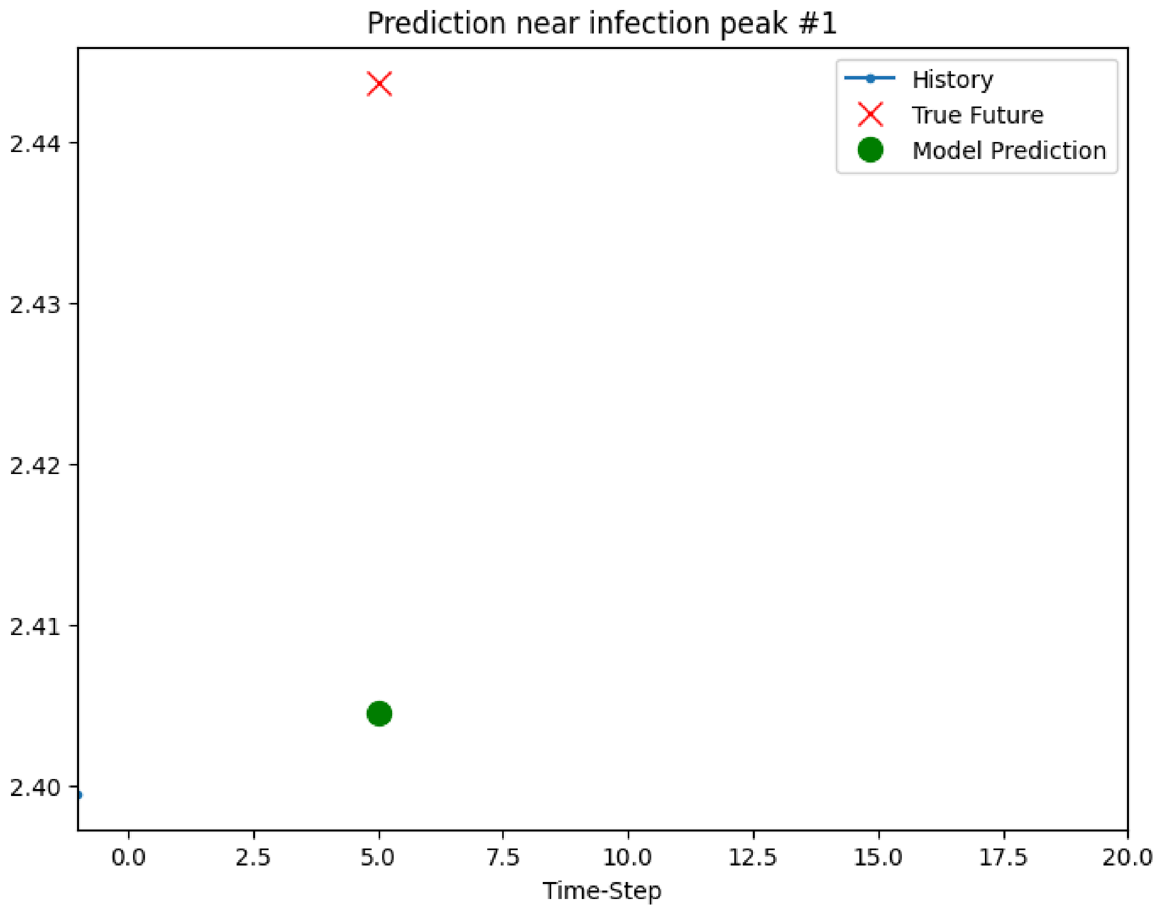
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step



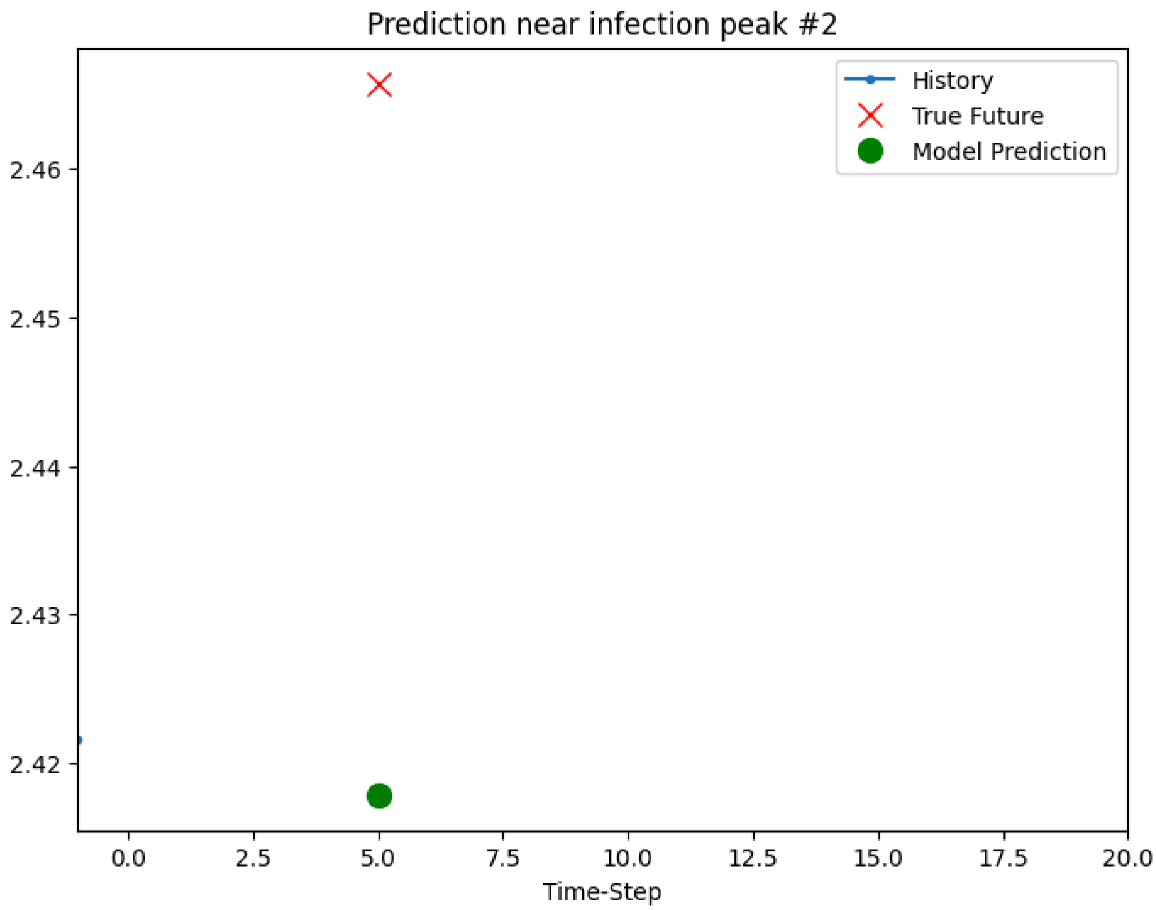
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step



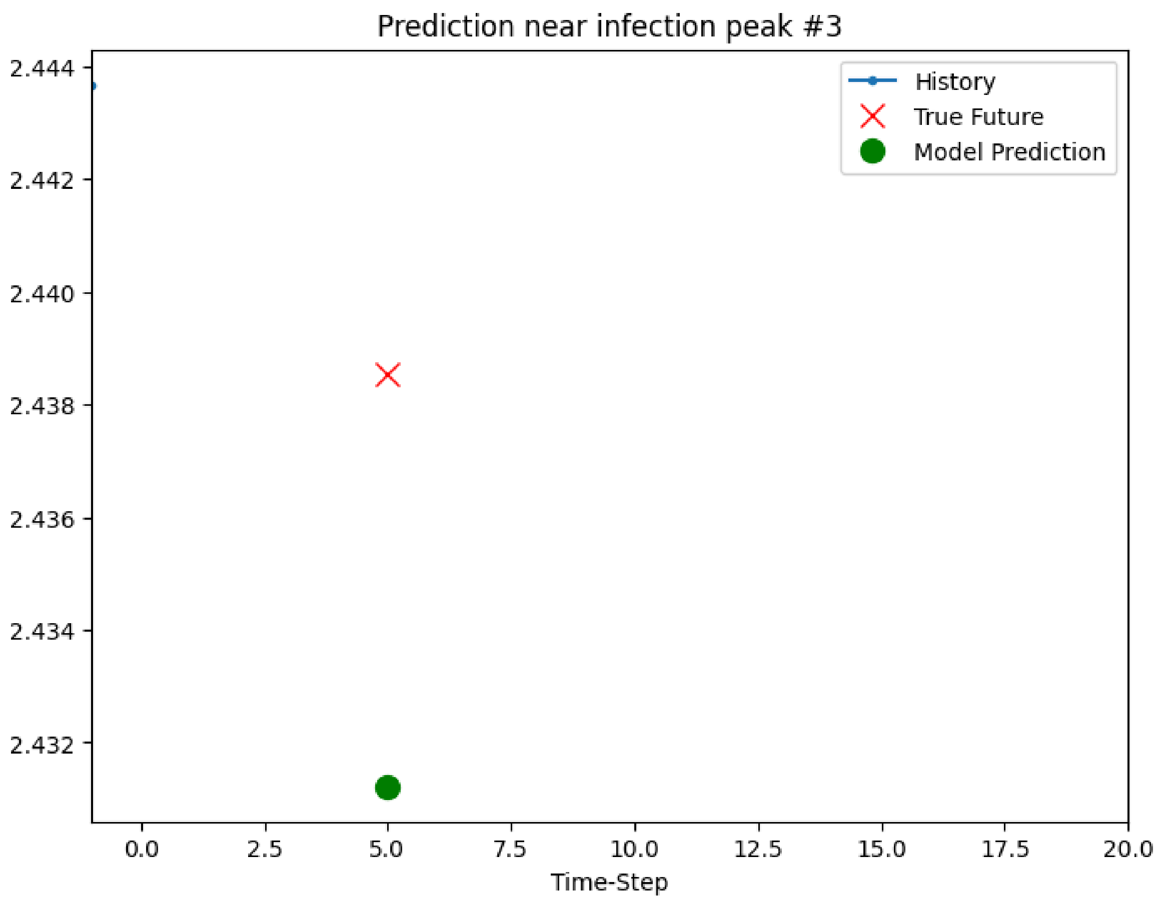

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 229ms/step
Peak #1 — Index: 142, True: 2.444, Predicted: 2.405
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step
Peak #2 — Index: 143, True: 2.466, Predicted: 2.418
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step
Peak #3 — Index: 144, True: 2.439, Predicted: 2.431
```

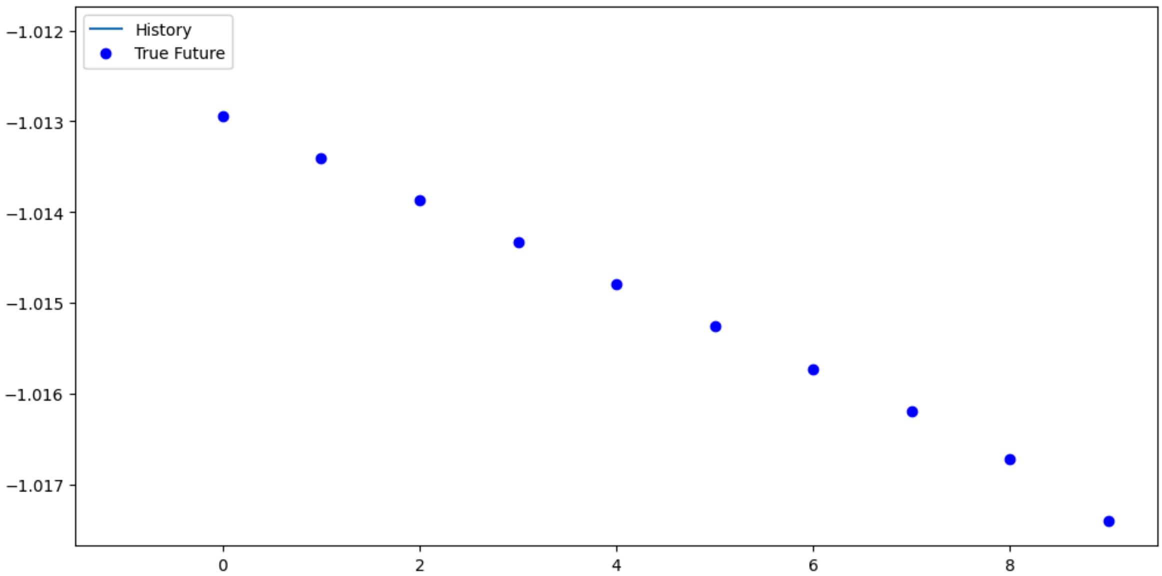


Multivariate GRU - Multiple Steps

In [50]:
```
past_history = 2
future_target = 10
STEP =2

x_train_multi, y_train_multi = multivariate_data(dataset, dfInfected_data, 0, TR
                                                 past_history, future_target,

x_val_multi, y_val_multi = multivariate_data(dataset, dfInfected_data, TRAIN_SPL
                                             past_history, future_target, STE

print ('Single window of past history : {}'.format(x_train_multi[0].shape))
print ('\nTarget window to predict : {}'.format(y_train_multi[0].shape))
print ('\nNumber of traing data points: {}'.format(x_train_multi.shape[0]))
print ('\nNumber of test data points: {}'.format(x_val_multi.shape[0]))

train_data_multi = tf.data.Dataset.from_tensor_slices((x_train_multi, y_train_mu
train_data_multi = train_data_multi.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZ

val_data_multi = tf.data.Dataset.from_tensor_slices((x_val_multi, y_val_multi))
val_data_multi = val_data_multi.batch(BATCH_SIZE).repeat()

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[:, 0]), label='History')
    plt.plot(np.arange(num_out), np.array(true_future), 'bo', label='True Future
    if prediction.any():
        plt.plot(np.arange(num_out), np.array(prediction), 'ro', label='Predicte
    plt.legend(loc='upper left')
    plt.show()

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

multi_step_model_GRU = tf.keras.models.Sequential()
multi_step_model_GRU.add(tf.keras.layers.GRU(32,
                                          return_sequences=True,
                                          input_shape=x_train_multi.shape[-2:]))
multi_step_model_GRU.add(tf.keras.layers.GRU(16, activation='relu'))
multi_step_model_GRU.add(tf.keras.layers.Dense(future_target))

multi_step_model_GRU.compile(optimizer=tf.keras.optimizers.RMSprop(clipvalue=1.0
multi_step_model_GRU.summary()
x_train_multi.shape[-2:]

for x, y in val_data_multi.take(1):
    print (multi_step_model.predict(x).shape)

multi_step_history_GRU = multi_step_model_GRU.fit(train_data_multi, epochs=EPOCH
                                          steps_per_epoch=EVALUATION_INTERVAL,
                                          validation_data=val_data_multi,
                                          validation_steps=50)

plot_train_history(multi_step_history_GRU, 'Multi-Step Training and validation l

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

```
# “Show diagrams with predictions around the peak of infection”

# Use dfInfected_data for peak detection
flat_data = dfInfected_data[0] if len(dfInfected_data.shape) == 2 else dfInfecte

# Find top 3 peak indices after enough history
peak_indices = np.argpartition(flat_data[past_history:], -3)[-3:] + past_history
peak_indices = sorted(peak_indices)

for i, peak_idx in enumerate(peak_indices):
    # Compute input indices based on history size and STEP
    input_indices = range(peak_idx - past_history * STEP, peak_idx, STEP)
    one = dfInfected_data[0][input_indices]
    two = dfRecovered_data[0][input_indices]
    three = dfDead_data[0][input_indices]

    input_seq = np.transpose(np.array([one, two, three])).reshape(1, -1, 3)

    # Predict the 10-step future
    prediction = multi_step_model_GRU.predict(input_seq)[0]
    true_future = flat_data[peak_idx:peak_idx + future_target]

    # If peak is too close to the end, skip
    if len(true_future) < future_target:
        continue

    print(f"Peak #{i+1} — index: {peak_idx}")
    print(f"True future: {true_future}")
    print(f"Predicted:    {prediction}")

    # Use your multi-step plot function
    multi_step_plot(input_seq[0], true_future, prediction)
```

```
Single window of past history : (1, 3)

Target window to predict : (10,)

Number of traing data points: 66015

Number of test data points: 7335
```



Model: "sequential_5"

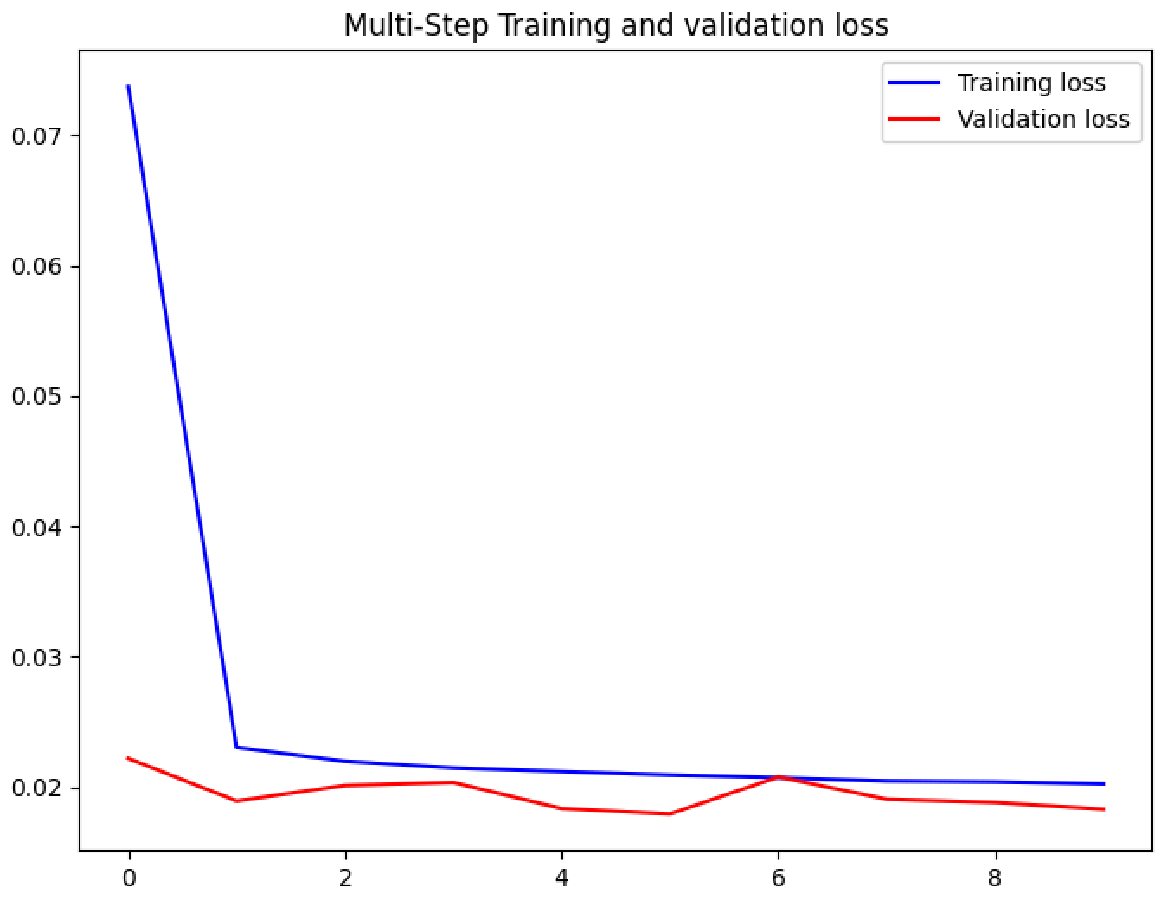
Layer (type)	Output Shape	Param #
gru_2 (GRU)	(None, 1, 32)	3,552
gru_3 (GRU)	(None, 16)	2,400
dense_7 (Dense)	(None, 10)	170

Total params: 6,122 (23.91 KB)

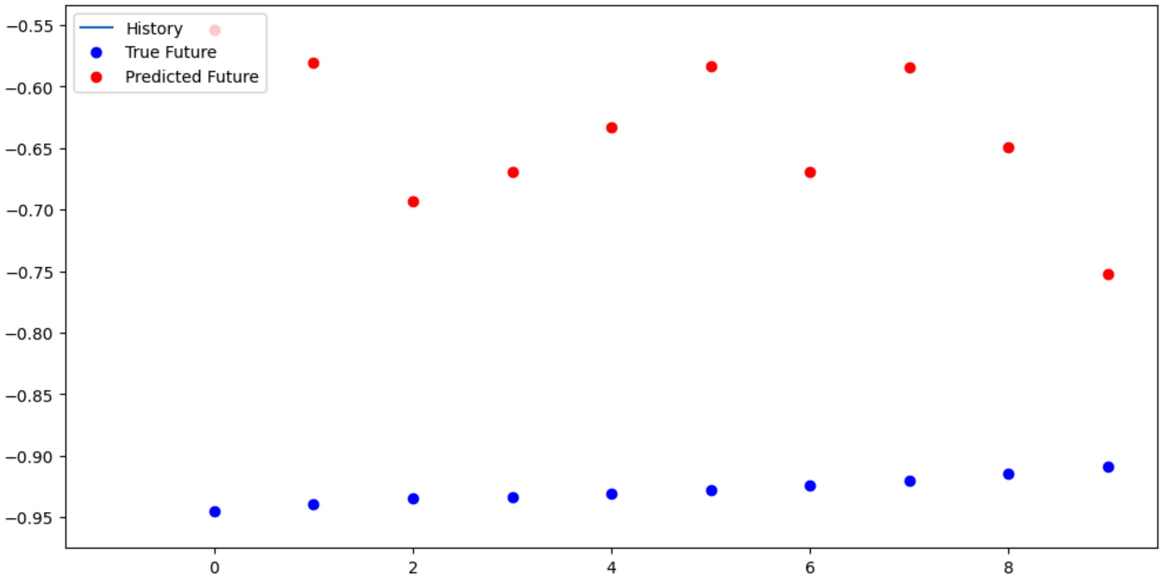
Trainable params: 6,122 (23.91 KB)

Non-trainable params: 0 (0.00 B)

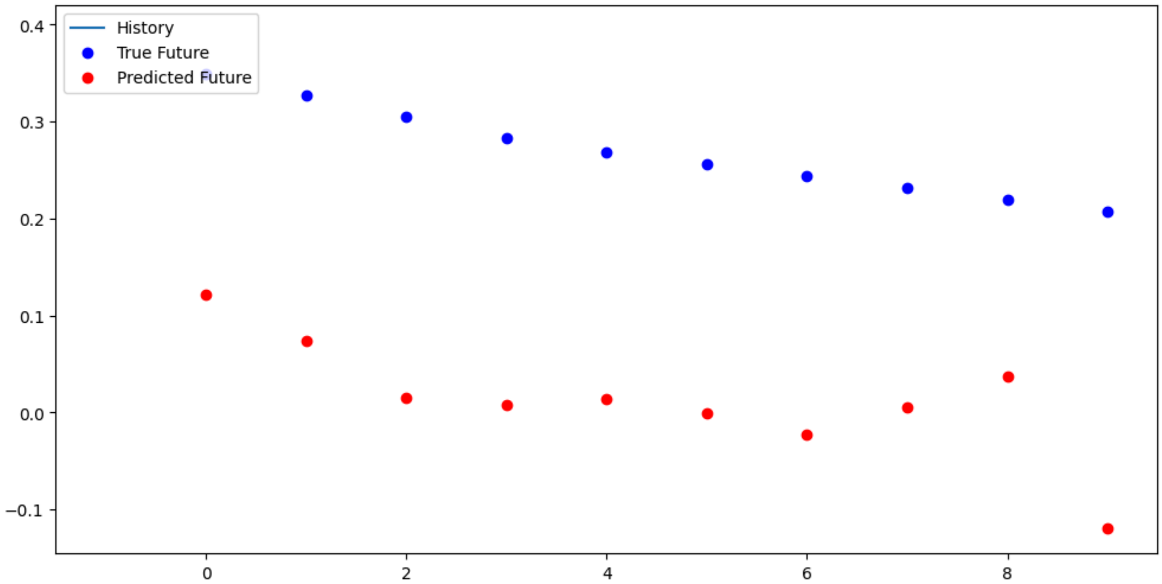
```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
(256, 10)
Epoch 1/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 10s 3ms/step - loss: 0.2011 - val_loss: 0.0222
Epoch 2/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0236 - val_loss: 0.0189
Epoch 3/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0221 - val_loss: 0.0201
Epoch 4/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0215 - val_loss: 0.0203
Epoch 5/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0213 - val_loss: 0.0183
Epoch 6/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0210 - val_loss: 0.0179
Epoch 7/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0207 - val_loss: 0.0208
Epoch 8/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0205 - val_loss: 0.0190
Epoch 9/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0206 - val_loss: 0.0188
Epoch 10/10
2000/2000 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - loss: 0.0202 - val_loss: 0.0183
```



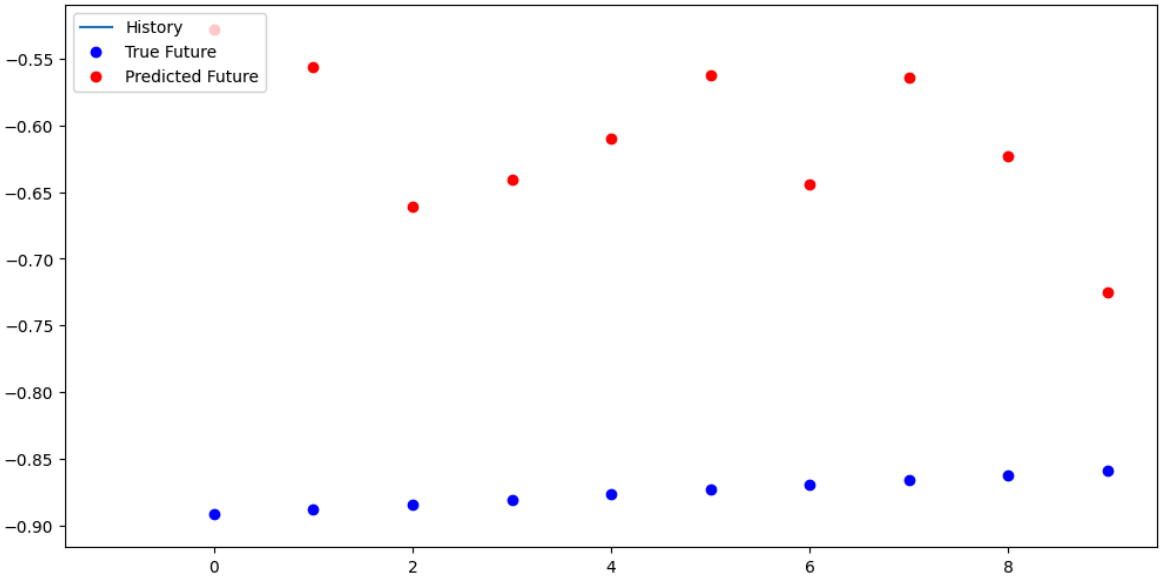
8/8 0s 5ms/step



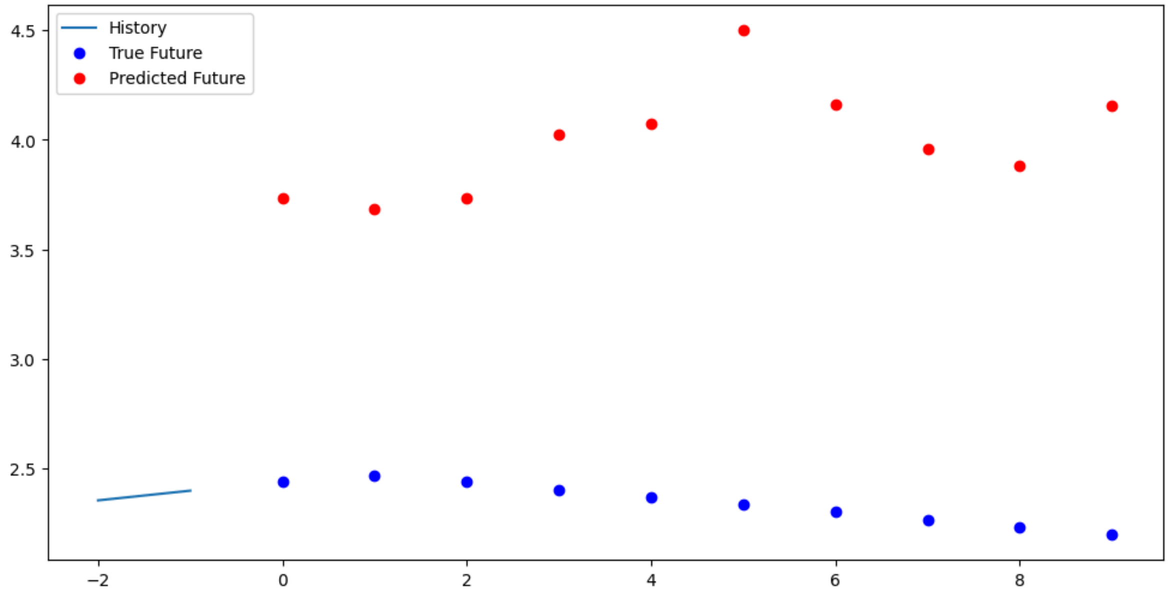
8/8 0s 5ms/step



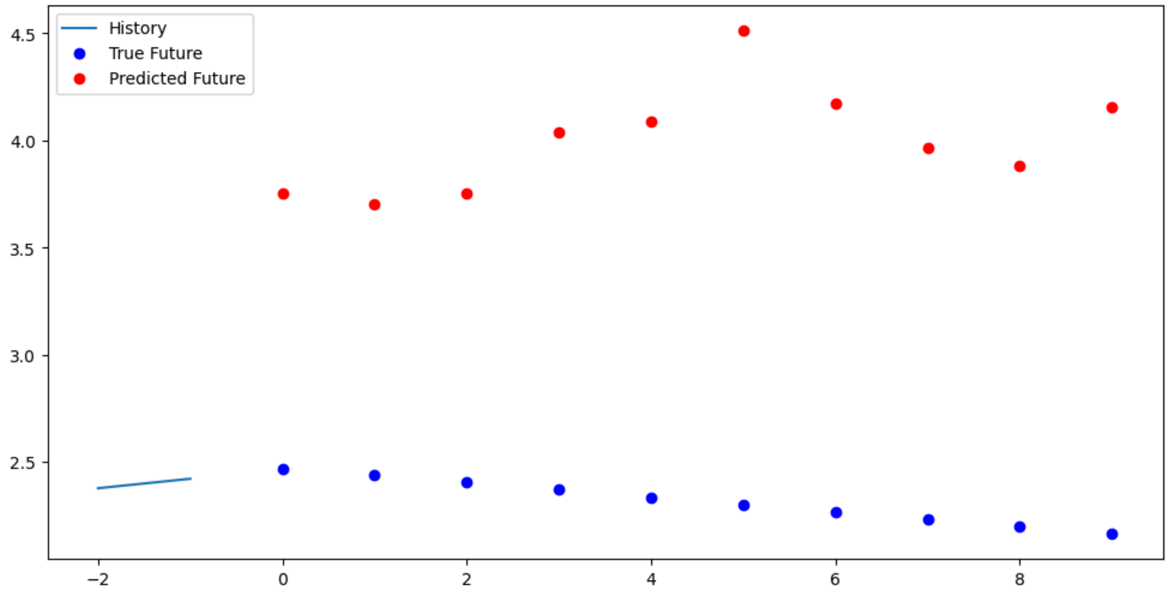

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
```



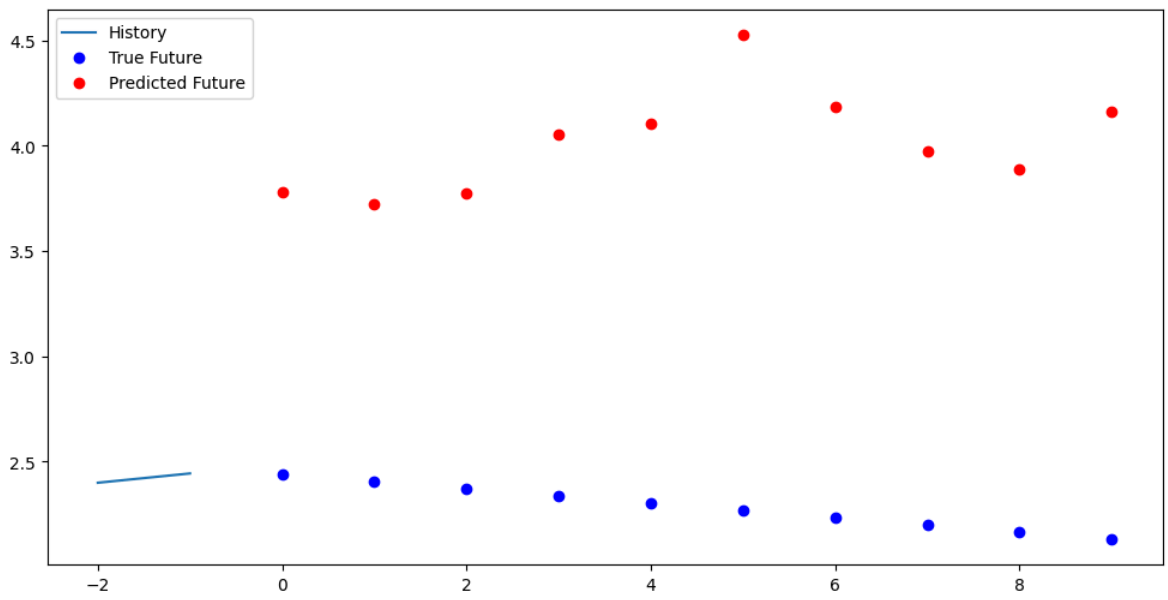

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 403ms/step
Peak #1 — index: 142
True future: [2.44368116 2.46575886 2.43853705 2.40422225 2.36990746 2.33559267
 2.30127788 2.26696308 2.23264829 2.1983335 ]
Predicted:    [3.7332795 3.684492  3.7363071 4.0239763 4.076452  4.4996624 4.1615
94
 3.9596968 3.88127   4.1548986]
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step
Peak #2 — index: 143
True future: [2.46575886 2.43853705 2.40422225 2.36990746 2.33559267 2.30127788
 2.26696308 2.23264829 2.1983335  2.16401871]
Predicted:    [3.7559133 3.7033772 3.7538433 4.0386944 4.0911064 4.512268  4.1728
16
 3.9660878 3.8841128 4.156949 ]
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 54ms/step
Peak #3 — index: 144
True future: [2.43853705 2.40422225 2.36990746 2.33559267 2.30127788 2.26696308
 2.23264829 2.1983335  2.16401871 2.12970391]
Predicted:    [3.778944  3.7228098 3.7722185 4.0543356 4.107062  4.5262184 4.1859
016
 3.974461  3.8891835 4.1617174]
```



History
True Future
Predicted Future
4.5
4.0
3.5
3.0
2.5
−2
0
2
4
6
8