```
plt.title("RNN vs. Original series with normal scale", fontsize=12)
plt.plot(data_train_inverse, "b--", markersize=10, label="Original series")
plt.plot(rnn_data_inverse, "g--", markersize=10, label="RNN generated
series")
plt.legend(loc="upper left")
plt.xlabel("Time")
plt.show()
```

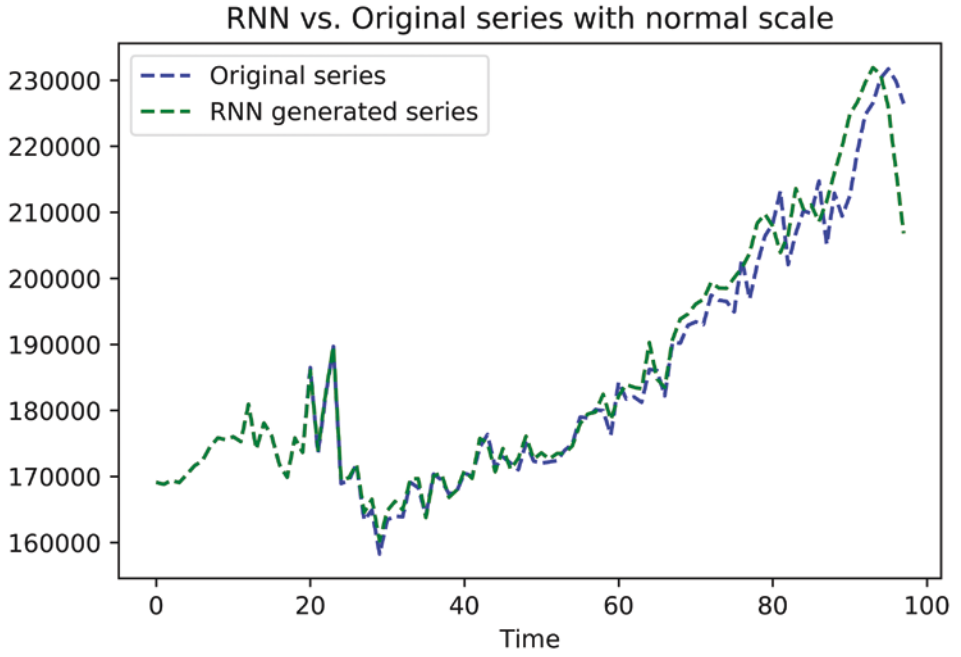

Figure 36-22. *Original series vs. RNN generated series – normal data values*

From the Keras LSTM code listing, the method **tf.keras.layers.LSTM()** is used to implement the LSTM recurrent layer. The attribute **return_sequences** is set to **True** to return the last output in the output sequence, or the full sequence.

RNN with TensorFlow 2.0: Multivariate Timeseries

The dataset for this example is the Dow Jones Index Data Set from the famous UCI Machine Learning Repository. In this stock dataset, each row contains the stock price record for a week including the percentage of return that stock has in the following week

percent_change_next_weeks_price(). For this example, the record for the previous week is used to predict the percent change in price for the next 2 weeks for Bank of America, BAC stock prices.

The method named **clean_dataset()** carries out some rudimentary cleanup of the dataset to make it suitable for modeling. The actions taken on this particular dataset involve removing the dollar sign from certain of the data columns, removing missing values, and rearranging the data columns so target attribute **percent_change_next_weeks_price** is the last column.

The method named **data_transform()** subselects the stock records belonging to 'Bank of America,' and the target attribute is adjusted so that the previous week record is used to predict the percent change in price for the next 2 weeks. Also, the dataset is split into training and testing sets. The method named **normalize_and_scale()** removes the non-numeric columns and scales the dataset attributes.

Again, the model will train on a GPU instance. The model will be a stacked GRU with multiple GRU layers. This stacking of RNN layers with memory cells makes the network more expressive and can learn more complex long-running sequences. If running the code on Google Colab, change the runtime type to GPU and install TensorFlow 2.0 with GPU package. The output plot in Figure 36-23 is the model predictions showing the targets and the lag training instances.

```
# import TensorFlow 2.0 with GPU
!pip install -q tf-nightly-gpu-2.0-preview

# import packages
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# confirm tensorflow can see GPU
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

```
# data file path
file_path = "dow_jones_index.data"

# load data
data = pd.read_csv(file_path, parse_dates=['date'], index_col='date')

# print column name
data.columns

# print column datatypes
data.dtypes

# parameters
outputs = 1
stock ='BAC'  # Bank of America

def clean_dataset(data):
    # strip dollar sign from `object` type columns
    col = ['open', 'high', 'low', 'close', 'next_weeks_open', 'next_weeks_
    close']
    data[col] = data[col].replace({'\$': ''}, regex=True)
    # drop NaN
    data.dropna(inplace=True)
    # rearrange columns
    columns = ['quarter', 'stock', 'open', 'high', 'low', 'close', 'volume',
       'percent_change_price', 'percent_change_volume_over_last_wk',
       'previous_weeks_volume', 'next_weeks_open', 'next_weeks_close',
       'days_to_next_dividend', 'percent_return_next_dividend',
       'percent_change_next_weeks_price']
    data = data[columns]
    return data

def data_transform(data):
    # select stock data belonging to Bank of America
    data = data[data.stock == stock]
    # adjust target(t) to depend on input (t-1)
    data.percent_change_next_weeks_price = data.percent_change_next_weeks_
    price.shift(-1)
```

```
    # remove nans as a result of the shifted values
    data = data.iloc[:-1,:]
    # split quarter 1 as training data and quarter 2 as testing data
    train_df = data[data.quarter == 1]
    test_df = data[data.quarter == 2]
    return (np.array(train_df), np.array(test_df))

def normalize_and_scale(train_df, test_df):
    # remove string columns and convert to float
    train_df = train_df[:,2:].astype(float,copy=False)
    test_df = test_df[:,2:].astype(float,copy=False)
    # MinMaxScaler - center and scale the dataset
    scaler = MinMaxScaler(feature_range=(0, 1))
    train_df_scale = scaler.fit_transform(train_df[:,2:])
    test_df_scale = scaler.fit_transform(test_df[:,2:])
    return (scaler, train_df_scale, test_df_scale)

# clean the dataset
data = clean_dataset(data)

# select Dow Jones stock and split into training and test sets
train_df, test_df = data_transform(data)

# scale the data
scaler, train_df_scaled, test_df_scaled = normalize_and_scale(train_df,
test_df)

# split train/ test
train_X, train_y = train_df_scaled[:, :-1], train_df_scaled[:, -1]
test_X, test_y = test_df_scaled[:, :-1], test_df_scaled[:, -1]

# reshape inputs to 3D array
train_X = train_X[:,None,:]
test_X = test_X[:,None,:]

# reshape outputs
train_y = np.reshape(train_y, (-1,outputs))
test_y = np.reshape(test_y, (-1,outputs))
```

```
# model parameters
batch_size = int(train_X.shape[0]/5)
length = train_X.shape[0]

# build model
model = tf.keras.Sequential()
model.add(tf.keras.layers.GRU(128, input_shape=train_X.shape[1:],
                              return_sequences=True))
model.add(tf.keras.layers.GRU(100, return_sequences=True))
model.add(tf.keras.layers.GRU(64))
model.add(tf.keras.layers.Dense(1))

# compile the model
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['mse'])

# print model summary
model.summary()

# create dataset pipeline
train_ds = tf.data.Dataset.from_tensor_slices(
    (train_X, train_y)).shuffle(len(train_X)).repeat().batch(batch_size)
test_ds = tf.data.Dataset.from_tensor_slices((test_X, test_y)).batch(batch_size)

# train the model
history = model.fit(train_ds, epochs=10,
                    steps_per_epoch=500)

# evaluate the model
loss, mse = model.evaluate(test_ds)

print('Test loss: {:.4f}'.format(loss))
print('Test mse: {:.4f}'.format(mse))

# predict
y_pred = model.predict(test_X)

# plot
plt.figure(1)
```

```
plt.title("Keras - GRU RNN Model Testing for '{}' stock".format(stock),
fontsize=12)
plt.plot(test_y, "g--", markersize=10, label="targets")
plt.plot(y_pred, "r--", markersize=10, label="model prediction")
plt.legend()
plt.xlabel("Time")
plt.show()
# plt.savefig('gru-bac-model.png', dpi=800)
```

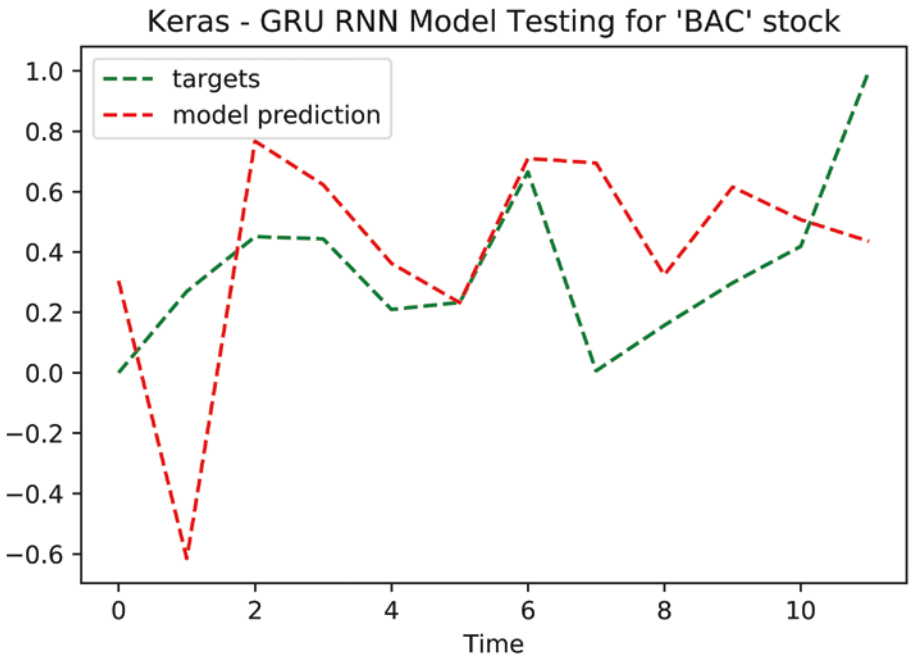

Figure 36-23. *GRU RNN Model Testing for Bank of America stock*

This chapter gave an overview of recurrent neural networks (RNNs) and its application in learning recurrent models for different types of sequence problems. The next chapter will discuss how we can use neural networks to reconstruct the inputs as some form of unsupervised learning using autoencoders.

CHAPTER 37

Autoencoders

Autoencoder is an unsupervised learning algorithm that uses neural networks to reconstruct the features of a dataset. Just like the unsupervised algorithms that we earlier discussed in the chapter on machine learning, autoencoders can be used to reduce the dimensionality of a dataset and to extract relevant features. Moreso, peculiar to autoencoders is the ability to generate more examples of the dataset after learning an internal representation (also called coding) that reconstructs the features of the inputs to the neural network.

An autoencoder receives as input the features of the dataset. These features are passed through a set of encoders, which are the hidden layers of a neural network to create an internal representation called codings. The learned coding is then used to reconstruct the output through a set of decoders, which are also hidden neural network layers. The autoencoder cannot merely do a trivial memorization of the inputs, because a constraint is placed on the encoders by reducing the input dimension to force the network to learn an efficient set of representation from which the decoders use to reconstruct the inputs.

Autoencoders with restricted Encoders and Decoders are called **undercomplete**. A reconstruction error term is used to evaluate the performance of an autoencoder by testing how well the output corresponds with the input. Of course, just like other neural networks, the neurons of the Encoders and Decoders have non-linear activation functions for learning complex patterns. An example of a simple autoencoder network architecture is shown in Figure 37-1.

E. Bisong, *Building Machine Learning and Deep Learning Models on Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-4470-8_37