When the outputs from these side-by-side networks are combined, it is easier to predict the next time step of a sequence having privy to the entire information gamut, because they process both information from the past and the future. Although this architecture was first designed for speech recognition tasks, it has performed impressively across a variety of other sequence prediction tasks. It is built to improve on the vanilla unidirectional LSTM which only has knowledge of the past.

This network is built on the understanding that some learning problems only make sense when a coherent set of information is present. For example, if a human interpreter is interpreting from one language to another, he first listens to a cohesive set of information in one language before interpreting to another language. This is because the context of an entire cohesive sentence gives the right basis for a correct interpretation.

RNN with TensorFlow 2.0: Univariate Timeseries

This section makes use of the Nigeria power consumption dataset to implement a univariate timeseries model with LSTM recurrent neural networks. The dataset for this example is the Nigeria power consumption data from January 1 to March 11 by Hipel and McLeod (1994), retrieved from DataMarket.

The dataset is preprocessed for timeseries modeling with RNNs by converting the data input and outputs into sequences using the method **'convert_to_ sequences'**. This method splits the dataset into rolling sequences consisting of 20 rows (or time steps) using a window of **1**}. In Figure 36-19, the example univariate dataset is converted into sequences of five time steps, where output sequence is one step ahead of the input sequence. Each sequence contains five rows (determined by the **time_steps** variable) and in this univariate case, 1 column.

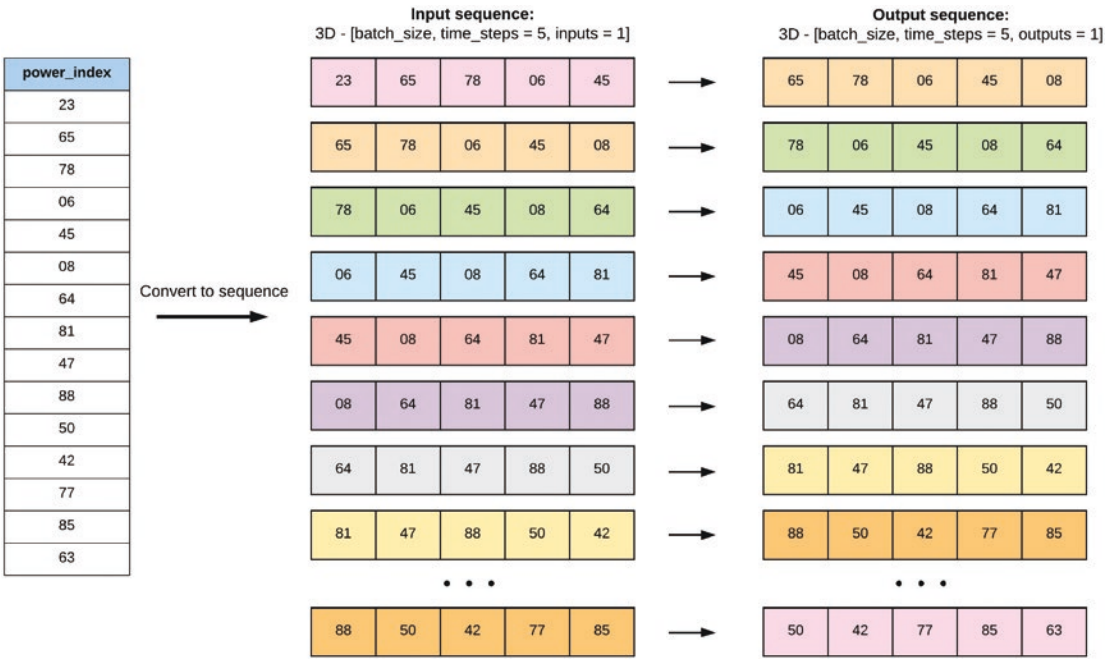

Figure 36-19. *Converting a univariate series into sequences for prediction with RNNs. Left: Sample univariate dataset. Center: Input sequence. Right: Output sequence*

When modeling using RNNs, it is important to scale the dataset to have values within the same range. The plot in Figure 36-20 shows predictions of the model along with the original targets and the lagging training instances. The next plots in Figure 36-21 and Figure 36-22 show the original series and the RNN generated series in both the scaled and normal values.

For increased training speed, the model will train on a GPU. If running the code on Google Colab, change the runtime type to GPU and install TensorFlow 2.0 with GPU package.

```
# 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 = "nigeria-power-consumption.csv"

# load data
parse_date = lambda dates: pd.datetime.strptime(dates, '%d-%m')
data = pd.read_csv(file_path, parse_dates=['Month'], index_col='Month',
                   date_parser=parse_date,
                   engine='python', skipfooter=2)

# print column name
data.columns

# change column names
data.rename(columns={'Nigeria power consumption': 'power-consumption'},
            inplace=True)

# split in training and evaluation set
data_train, data_eval = train_test_split(data, test_size=0.2,
shuffle=False)

# MinMaxScaler - center and scale the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
data_train = scaler.fit_transform(data_train)
data_eval = scaler.fit_transform(data_eval)

# adjust univariate data for timeseries prediction
def convert_to_sequences(data, sequence, is_target=False):
    temp_df = []
    for i in range(len(data) - sequence):
        if is_target:
```

```
            temp_df.append(data[(i+1): (i+1) + sequence])
        else:
            temp_df.append(data[i: i + sequence])
    return np.array(temp_df)

# parameters
time_steps = 20
batch_size = 50

# create training and testing data
train_x = convert_to_sequences(data_train, time_steps, is_target=False)
train_y = convert_to_sequences(data_train, time_steps, is_target=True)

eval_x = convert_to_sequences(data_eval, time_steps, is_target=False)
eval_y = convert_to_sequences(data_eval, time_steps, is_target=True)

# build model
model = tf.keras.Sequential()
model.add(tf.keras.layers.LSTM(128, input_shape=train_x.shape[1:],
                               return_sequences=True))
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((eval_x, eval_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(eval_x)

# plot predicted sequence
plt.title("Model Testing", fontsize=12)
plt.plot(eval_x[0,:,0], "b--", markersize=10, label="training instance")
plt.plot(eval_y[0,:,0], "g--", markersize=10, label="targets")
plt.plot(y_pred[0,:,0], "r--", markersize=10, label="model prediction")
plt.legend(loc="upper left")
plt.xlabel("Time")
plt.show()
```

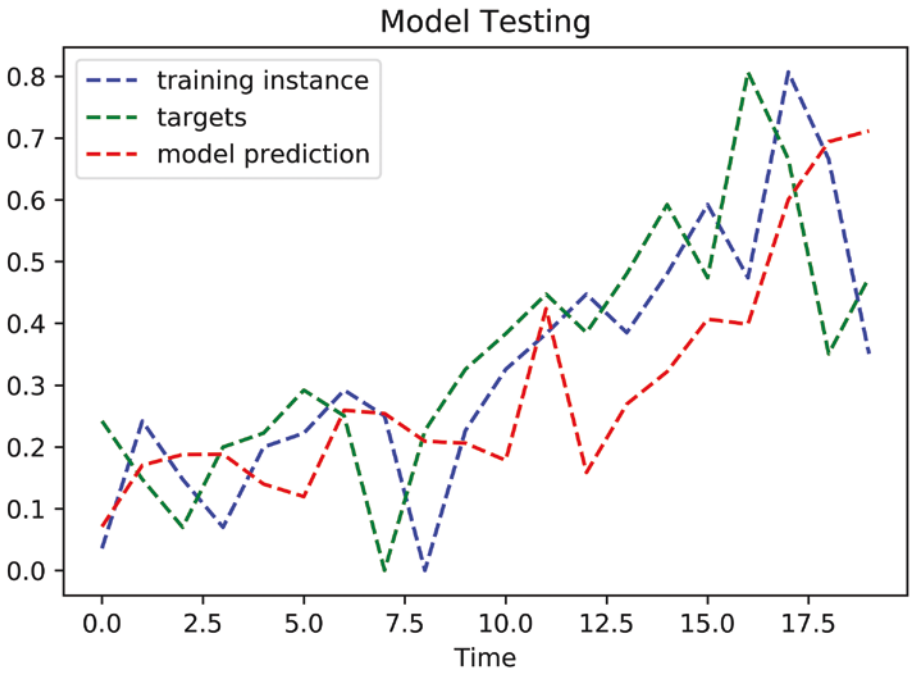

Figure 36-20. *Keras LSTM Model Testing*

```
# use model to predict sequences using training data as seed
rnn_data = list(data_train[:20])
for i in range(len(data_train) - time_steps):
    batch = np.array(rnn_data[-time_steps:]).reshape(1, time_steps, 1)
    y_pred = model.predict(batch)
    rnn_data.append(y_pred[0, -1, 0])

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

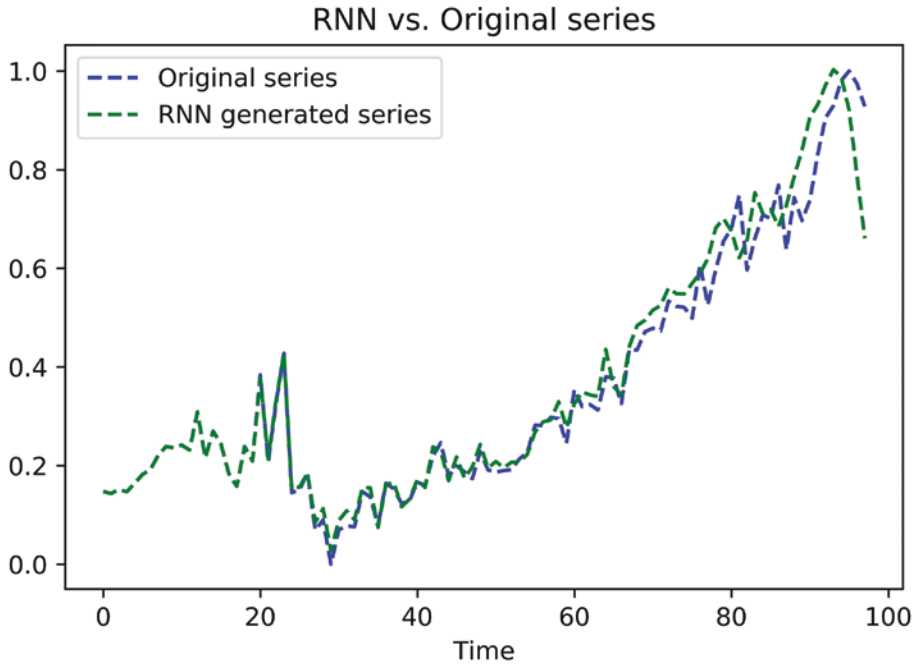

Figure 36-21. *Original series vs. RNN generated series – scaled data values*

```
# inverse to normal scale and plot
data_train_inverse = scaler.inverse_transform(data_train.reshape(-1, 1))
rnn_data_inverse = scaler.inverse_transform(np.array(rnn_data).reshape(-1, 1))
```

```
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()
```

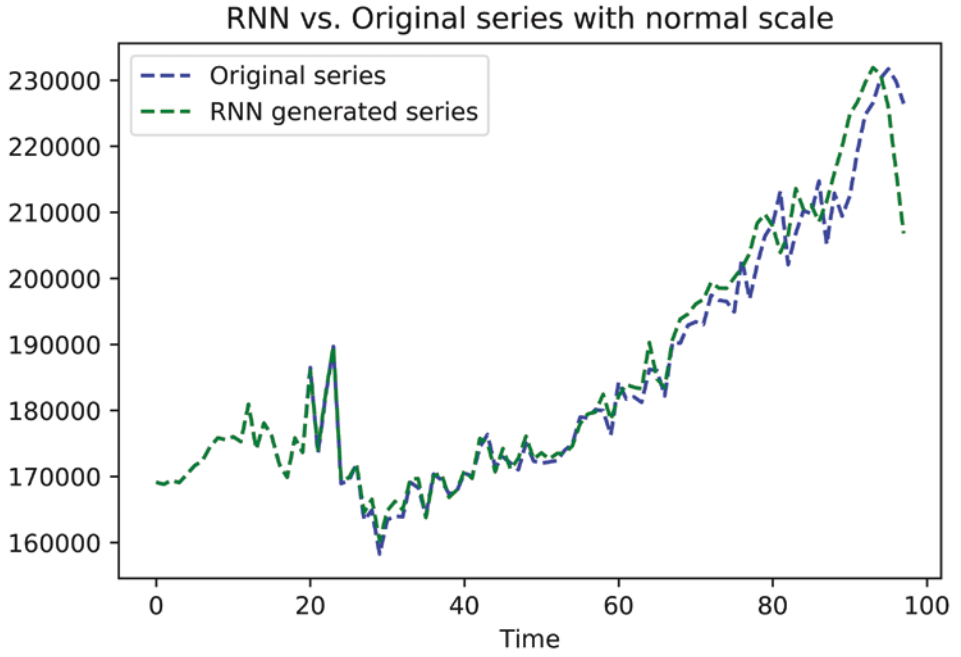

Figure 36-22. *Original series vs. RNN generated series – normal data values*

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