Ungraded Lab: Using a Simple RNN for forecasting

In this lab, you will start to use recurrent neural networks (RNNs) to build a forecasting model. In particular, you will:

- build a stacked RNN using `simpleRNN` layers
- use `Lambda` layers to reshape the input and scale the output
- use the Huber loss during training
- use batched data windows to generate model predictions

You will train this on the same synthetic dataset from last week so the initial steps will be the same. Let's begin!

Imports

In [1]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

Utilities

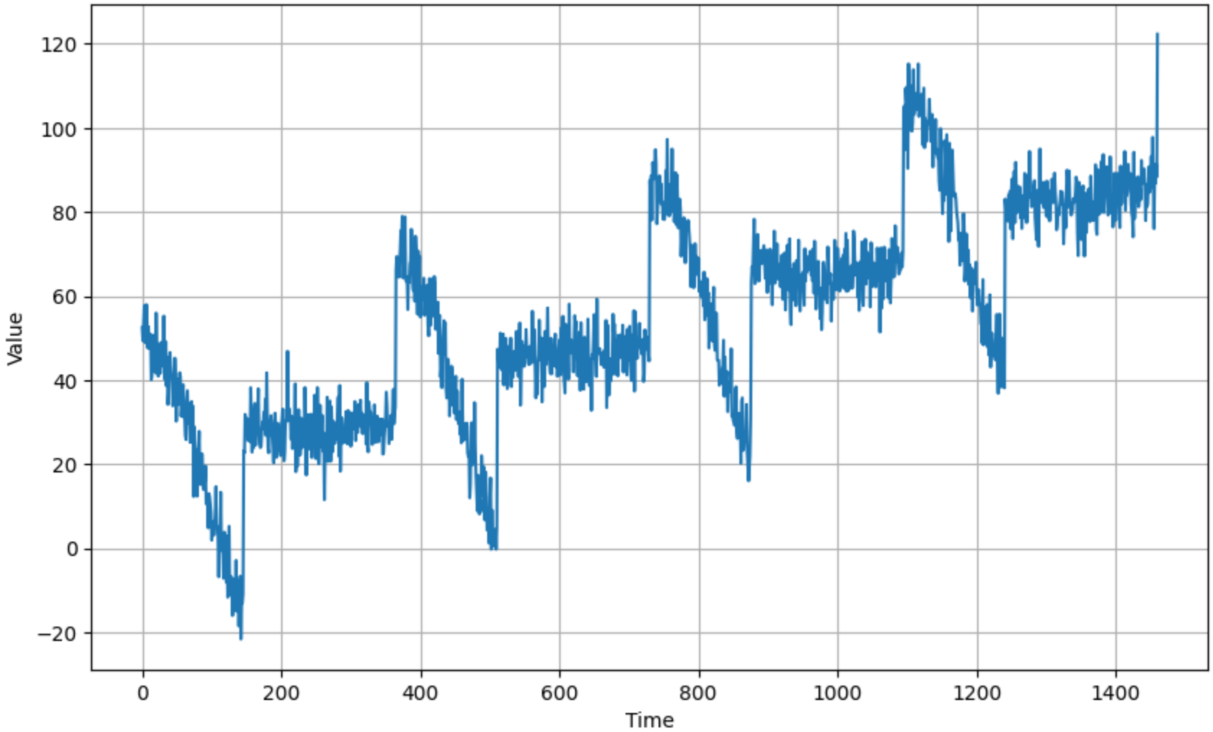
In [2]:
```
def plot_series(time, series, format="-", start=0, end=None):
    """
    Visualizes time series data

    Args:
      time (array of int) - contains the time steps
      series (array of int) - contains the measurements for each time step
      format - line style when plotting the graph
      start - first time step to plot
      end - last time step to plot
    """

    # Setup dimensions of the graph figure
    plt.figure(figsize=(10, 6))

    if type(series) is tuple:

      for series_num in series:
        # Plot the time series data
        plt.plot(time[start:end], series_num[start:end], format)

    else:
      # Plot the time series data
      plt.plot(time[start:end], series[start:end], format)

    # Label the x-axis
    plt.xlabel("Time")

    # Label the y-axis
    plt.ylabel("Value")

    # Overlay a grid on the graph
    plt.grid(True)

    # Draw the graph on screen
    plt.show()

def trend(time, slope=0):
    """
    Generates synthetic data that follows a straight line given a slope value.

    Args:
      time (array of int) - contains the time steps
      slope (float) - determines the direction and steepness of the line

    Returns:
      series (array of float) - measurements that follow a straight line
    """

    # Compute the linear series given the slope
```

```python
    series = slope * time

    return series

def seasonal_pattern(season_time):
    """
    Just an arbitrary pattern, you can change it if you wish

    Args:
      season_time (array of float) - contains the measurements per time step

    Returns:
      data_pattern (array of float) -  contains revised measurement values according
                                  to the defined pattern
    """

    # Generate the values using an arbitrary pattern
    data_pattern = np.where(season_time < 0.4,
                    np.cos(season_time * 2 * np.pi),
                    1 / np.exp(3 * season_time))

    return data_pattern

def seasonality(time, period, amplitude=1, phase=0):
    """
    Repeats the same pattern at each period

    Args:
      time (array of int) - contains the time steps
      period (int) - number of time steps before the pattern repeats
      amplitude (int) - peak measured value in a period
      phase (int) - number of time steps to shift the measured values

    Returns:
      data_pattern (array of float) - seasonal data scaled by the defined amplitude
    """

    # Define the measured values per period
    season_time = ((time + phase) % period) / period

    # Generates the seasonal data scaled by the defined amplitude
    data_pattern = amplitude * seasonal_pattern(season_time)

    return data_pattern

def noise(time, noise_level=1, seed=None):
    """Generates a normally distributed noisy signal

    Args:
      time (array of int) - contains the time steps
      noise_level (float) - scaling factor for the generated signal
      seed (int) - number generator seed for repeatability

    Returns:
      noise (array of float) - the noisy signal
    """

    # Initialize the random number generator
    rnd = np.random.RandomState(seed)

    # Generate a random number for each time step and scale by the noise level
    noise = rnd.randn(len(time)) * noise_level

    return noise
```

Generate the Synthetic Data

In [3]:
```python
# Parameters
time = np.arange(4 * 365 + 1, dtype="float32")
baseline = 10
amplitude = 40
slope = 0.05
noise_level = 5

# Create the series
series = baseline + trend(time, slope) + seasonality(time, period=365, amplitude=amplitude)
```

```
# Update with noise
series += noise(time, noise_level, seed=42)

# Plot the results
plot_series(time, series)
```



Split the Dataset

In [4]:
```
# Define the split time
split_time = 1000

# Get the train set
time_train = time[:split_time]
x_train = series[:split_time]

# Get the validation set
time_valid = time[split_time:]
x_valid = series[split_time:]
```

Prepare Features and Labels

In [5]:
```
# Parameters
window_size = 20
batch_size = 32
shuffle_buffer_size = 1000
```

You will be using `SimpleRNN` layers later and as mentioned in its documentation, these expect a 3-dimensional tensor input with the shape `[batch, timesteps, feature`]. With that, you need to reshape your window from `(32, 20)` to `(32, 20, 1)` . This means the 20 data points in the window will be mapped to 20 timesteps of the RNN. To implement this, you will add an `expand_dims()` to the `windowed_dataset()` function you used in the previous labs.

Note: Technically, you will only need this extra line if you don't specify the input shape as you will do later when you build the model. Nonetheless, it is best practice to define transformations like this, especially in data pipelines. It can help make debugging easier in case you have problems later on.

In [6]:
```
def windowed_dataset(series, window_size, batch_size, shuffle_buffer):
    """Generates dataset windows

    Args:
      series (array of float) - contains the values of the time series
```

```
      window_size (int) - the number of time steps to include in the feature
      batch_size (int) - the batch size
      shuffle_buffer(int) - buffer size to use for the shuffle method

    Returns:
      dataset (TF Dataset) - TF Dataset containing time windows
    """

    # Add an axis for the feature dimension of RNN layers
    series = tf.expand_dims(series, axis=-1)

    # Generate a TF Dataset from the series values
    dataset = tf.data.Dataset.from_tensor_slices(series)

    # Window the data but only take those with the specified size
    dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)

    # Flatten the windows by putting its elements in a single batch
    dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))

    # Create tuples with features and labels
    dataset = dataset.map(lambda window: (window[:-1], window[-1]))

    # Shuffle the windows
    dataset = dataset.shuffle(shuffle_buffer)

    # Create batches of windows
    dataset = dataset.batch(batch_size)

    # Optimize the dataset for training
    dataset = dataset.cache().prefetch(1)

    return dataset
```

In [7]:
```
# Generate the dataset windows
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)
```

In [8]:
```
# Print shapes of feature and label
for window in dataset.take(1):
  print(f'shape of feature: {window[0].shape}')
  print(f'shape of label: {window[1].shape}')
```

```
shape of feature: (32, 20, 1)
shape of label: (32, 1)
```

Build the Model

Your model is composed mainly of SimpleRNN layers. As mentioned in the lectures, this type of RNN simply routes its output back to the input. You will stack two of these layers in your model so the first one should have `return_sequences` set to `True` .

Normally, you can just have a `Dense` layer output as shown in the previous labs. However, you can help the training by scaling up the output to around the same figures as your labels. This will depend on the activation functions you used in your model. `SimpleRNN` uses *tanh* by default and that has an output range of `[-1,1]` . You will use a `Lambda` layer to scale the output by 100 before it adjusts the layer weights. `Lambda` layers can be a useful tool to experiment with simple transformations like this. Feel free to remove this layer later after this lab and see what results you get.

In [9]:
```
# Build the Model
model_tune = tf.keras.models.Sequential([
    tf.keras.Input(shape=(window_size, 1)),
    tf.keras.layers.SimpleRNN(40, return_sequences=True),
    tf.keras.layers.SimpleRNN(40),
    tf.keras.layers.Dense(1),
    tf.keras.layers.Lambda(lambda x: x * 100.0)
])

# Print the model summary
model_tune.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
simple_rnn (SimpleRNN)	(None, 20, 40)	1,680
simple_rnn_1 (SimpleRNN)	(None, 40)	3,240
dense (Dense)	(None, 1)	41
lambda (Lambda)	(None, 1)	0

Total params: 4,961 (19.38 KB)

Trainable params: 4,961 (19.38 KB)

Non-trainable params: 0 (0.00 B)

Tune the Learning Rate

You will then tune the learning rate as before. You will define a learning rate schedule that changes this hyperparameter dynamically. You will use the Huber Loss as your loss function to minimize sensitivity to outliers.

In [10]:

```
# Set the learning rate scheduler
lr_schedule = tf.keras.callbacks.LearningRateScheduler(
    lambda epoch: 1e-8 * 10**(epoch / 20))

# Initialize the optimizer
optimizer = tf.keras.optimizers.SGD(momentum=0.9)

# Set the training parameters
model_tune.compile(loss=tf.keras.losses.Huber(), optimizer=optimizer)

# Train the model
history = model_tune.fit(dataset, epochs=100, callbacks=[lr_schedule])
```

```
Epoch 1/100
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
I0000 00:00:1745440007.560499     234 service.cc:145] XLA service 0x7645190 initialized for platform CUDA (this does not gu
arantee that XLA will be used). Devices:
I0000 00:00:1745440007.560621     234 service.cc:153]   StreamExecutor device (0): NVIDIA A10G, Compute Capability 8.6
     16/Unknown 2s 3ms/step - loss: 120.8628
I0000 00:00:1745440008.257535     234 device_compiler.h:188] Compiled cluster using XLA!  This line is logged at most once
for the lifetime of the process.
```

```
31/31 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - loss: 120.1881 - learning_rate: 1.0000e-08
Epoch 2/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 115.8633 - learning_rate: 1.1220e-08
Epoch 3/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 110.1585 - learning_rate: 1.2589e-08
Epoch 4/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 103.6906 - learning_rate: 1.4125e-08
Epoch 5/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 96.5467 - learning_rate: 1.5849e-08
Epoch 6/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 88.8289 - learning_rate: 1.7783e-08
Epoch 7/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 80.6533 - learning_rate: 1.9953e-08
Epoch 8/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 72.1278 - learning_rate: 2.2387e-08
Epoch 9/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 63.3099 - learning_rate: 2.5119e-08
Epoch 10/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 54.2411 - learning_rate: 2.8184e-08
Epoch 11/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 45.0442 - learning_rate: 3.1623e-08
Epoch 12/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 36.1606 - learning_rate: 3.5481e-08
Epoch 13/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 28.3809 - learning_rate: 3.9811e-08
Epoch 14/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 22.9959 - learning_rate: 4.4668e-08
Epoch 15/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 19.8783 - learning_rate: 5.0119e-08
Epoch 16/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 18.0568 - learning_rate: 5.6234e-08
Epoch 17/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.2118 - learning_rate: 6.3096e-08
Epoch 18/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.9576 - learning_rate: 7.0795e-08
Epoch 19/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.8316 - learning_rate: 7.9433e-08
Epoch 20/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.7078 - learning_rate: 8.9125e-08
Epoch 21/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.5697 - learning_rate: 1.0000e-07
Epoch 22/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.4144 - learning_rate: 1.1220e-07
Epoch 23/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.2414 - learning_rate: 1.2589e-07
Epoch 24/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.0460 - learning_rate: 1.4125e-07
Epoch 25/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 15.8264 - learning_rate: 1.5849e-07
Epoch 26/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 15.5815 - learning_rate: 1.7783e-07
Epoch 27/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 15.2942 - learning_rate: 1.9953e-07
Epoch 28/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.9549 - learning_rate: 2.2387e-07
Epoch 29/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.5699 - learning_rate: 2.5119e-07
Epoch 30/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.1194 - learning_rate: 2.8184e-07
Epoch 31/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 13.5730 - learning_rate: 3.1623e-07
Epoch 32/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.8981 - learning_rate: 3.5481e-07
Epoch 33/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.0640 - learning_rate: 3.9811e-07
Epoch 34/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 11.0179 - learning_rate: 4.4668e-07
Epoch 35/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.8723 - learning_rate: 5.0119e-07
Epoch 36/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.1938 - learning_rate: 5.6234e-07
Epoch 37/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.8749 - learning_rate: 6.3096e-07
Epoch 38/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.6374 - learning_rate: 7.0795e-07
Epoch 39/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.4514 - learning_rate: 7.9433e-07
Epoch 40/100
```

```
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.2954 - learning_rate: 8.9125e-07
Epoch 41/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.2237 - learning_rate: 1.0000e-06
Epoch 42/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.3001 - learning_rate: 1.1220e-06
Epoch 43/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.3742 - learning_rate: 1.2589e-06
Epoch 44/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.0223 - learning_rate: 1.4125e-06
Epoch 45/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.4104 - learning_rate: 1.5849e-06
Epoch 46/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.1494 - learning_rate: 1.7783e-06
Epoch 47/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.2208 - learning_rate: 1.9953e-06
Epoch 48/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.6214 - learning_rate: 2.2387e-06
Epoch 49/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.6125 - learning_rate: 2.5119e-06
Epoch 50/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.5996 - learning_rate: 2.8184e-06
Epoch 51/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.4984 - learning_rate: 3.1623e-06
Epoch 52/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.3712 - learning_rate: 3.5481e-06
Epoch 53/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.5584 - learning_rate: 3.9811e-06
Epoch 54/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.4878 - learning_rate: 4.4668e-06
Epoch 55/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.5475 - learning_rate: 5.0119e-06
Epoch 56/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.2370 - learning_rate: 5.6234e-06
Epoch 57/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.9100 - learning_rate: 6.3096e-06
Epoch 58/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.8719 - learning_rate: 7.0795e-06
Epoch 59/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.4626 - learning_rate: 7.9433e-06
Epoch 60/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 10.6782 - learning_rate: 8.9125e-06
Epoch 61/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.3014 - learning_rate: 1.0000e-05
Epoch 62/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.7094 - learning_rate: 1.1220e-05
Epoch 63/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 11.2428 - learning_rate: 1.2589e-05
Epoch 64/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.0759 - learning_rate: 1.4125e-05
Epoch 65/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 10.2192 - learning_rate: 1.5849e-05
Epoch 66/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.7364 - learning_rate: 1.7783e-05
Epoch 67/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.6304 - learning_rate: 1.9953e-05
Epoch 68/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.6839 - learning_rate: 2.2387e-05
Epoch 69/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.0312 - learning_rate: 2.5119e-05
Epoch 70/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.0763 - learning_rate: 2.8184e-05
Epoch 71/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.7350 - learning_rate: 3.1623e-05
Epoch 72/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.5179 - learning_rate: 3.5481e-05
Epoch 73/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 10.5622 - learning_rate: 3.9811e-05
Epoch 74/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.0263 - learning_rate: 4.4668e-05
Epoch 75/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.0688 - learning_rate: 5.0119e-05
Epoch 76/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.5379 - learning_rate: 5.6234e-05
Epoch 77/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.3152 - learning_rate: 6.3096e-05
Epoch 78/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.4009 - learning_rate: 7.0795e-05
Epoch 79/100
```

```
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.2292 - learning_rate: 7.9433e-05
Epoch 80/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.4689 - learning_rate: 8.9125e-05
Epoch 81/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.9852 - learning_rate: 1.0000e-04
Epoch 82/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.8198 - learning_rate: 1.1220e-04
Epoch 83/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 15.6130 - learning_rate: 1.2589e-04
Epoch 84/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.4930 - learning_rate: 1.4125e-04
Epoch 85/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 21.4393 - learning_rate: 1.5849e-04
Epoch 86/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 14.8859 - learning_rate: 1.7783e-04
Epoch 87/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.7027 - learning_rate: 1.9953e-04
Epoch 88/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.0434 - learning_rate: 2.2387e-04
Epoch 89/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 18.4220 - learning_rate: 2.5119e-04
Epoch 90/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.3628 - learning_rate: 2.8184e-04
Epoch 91/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.2993 - learning_rate: 3.1623e-04
Epoch 92/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.8953 - learning_rate: 3.5481e-04
Epoch 93/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.5378 - learning_rate: 3.9811e-04
Epoch 94/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 18.0501 - learning_rate: 4.4668e-04
Epoch 95/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.9083 - learning_rate: 5.0119e-04
Epoch 96/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 15.4231 - learning_rate: 5.6234e-04
Epoch 97/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.0557 - learning_rate: 6.3096e-04
Epoch 98/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.3949 - learning_rate: 7.0795e-04
Epoch 99/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 17.5468 - learning_rate: 7.9433e-04
Epoch 100/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 19.9231 - learning_rate: 8.9125e-04
```

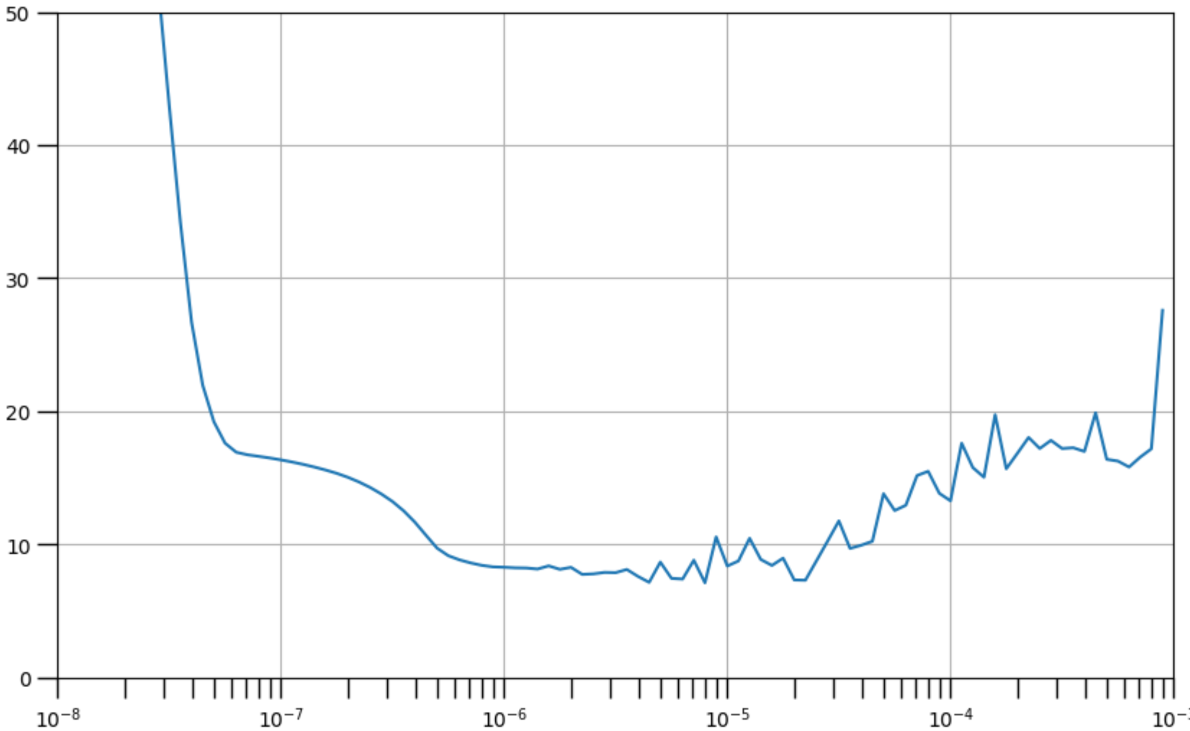
You can visualize the results and pick an optimal learning rate.

In [11]:

```python
# Define the learning rate array
lrs = 1e-8 * (10 ** (np.arange(100) / 20))

# Set the figure size
plt.figure(figsize=(10, 6))

# Set the grid
plt.grid(True)

# Plot the loss in log scale
plt.semilogx(lrs, history.history["loss"])

# Increase the tickmarks size
plt.tick_params('both', length=10, width=1, which='both')

# Set the plot boundaries
plt.axis([1e-8, 1e-3, 0, 50])
```

Out[11]:

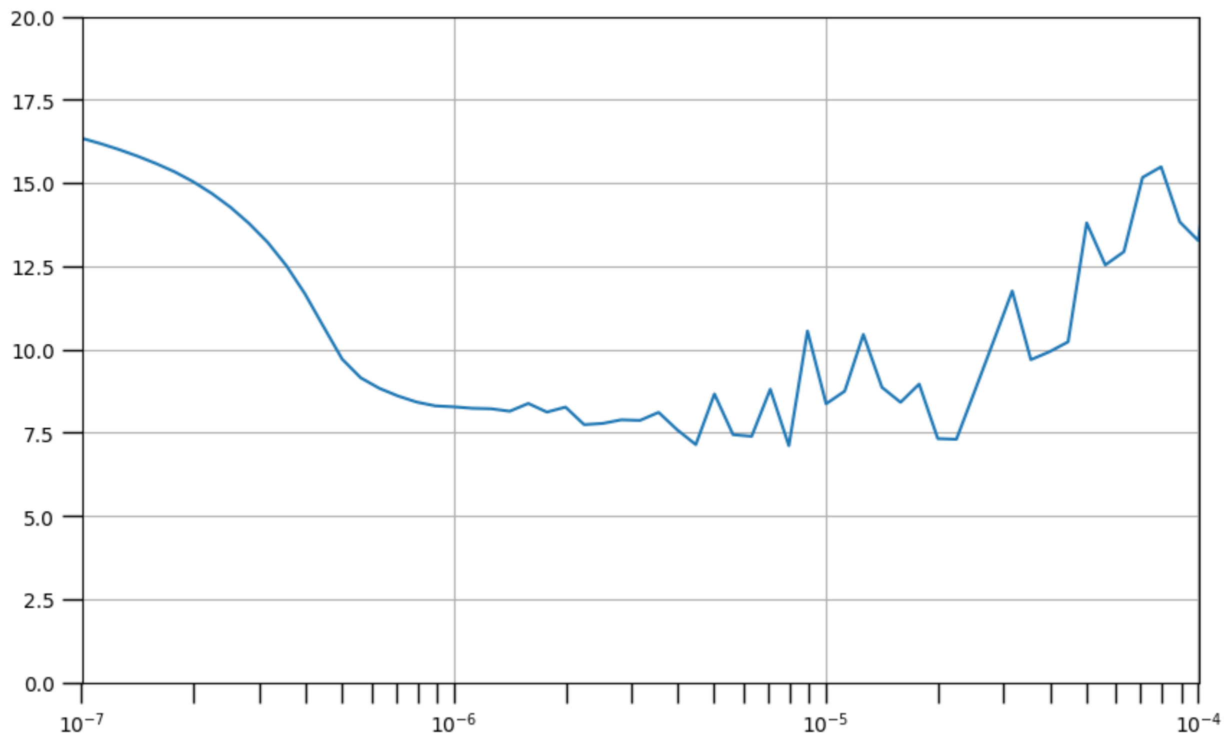
```
(1e-08, 0.001, 0.0, 50.0)
```



You can change the boundaries of the graph if you want to zoom in. The cell below chooses a narrower range so you can see more clearly where the graph becomes unstable.

In [12]:
```python
# Set the figure size
plt.figure(figsize=(10, 6))

# Set the grid
plt.grid(True)

# Plot the loss in log scale
plt.semilogx(lrs, history.history["loss"])

# Increase the tickmarks size
plt.tick_params('both', length=10, width=1, which='both')

# Set the plot boundaries
plt.axis([1e-7, 1e-4, 0, 20])
```

Out[12]:
```
(1e-07, 0.0001, 0.0, 20.0)
```



Train the Model

You can then declare the model again and train with the learning rate you picked. It is set to `1e-6` by default but feel free to change it.

In [13]:
```
# Build the model
model = tf.keras.models.Sequential([
    tf.keras.Input(shape=(window_size,1)),
    tf.keras.layers.SimpleRNN(40, return_sequences=True),
    tf.keras.layers.SimpleRNN(40),
    tf.keras.layers.Dense(1),
    tf.keras.layers.Lambda(lambda x: x * 100.0)
])

# Set the learning rate
learning_rate = 1e-6

# Set the optimizer
optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate, momentum=0.9)

# Set the training parameters
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])

# Train the model
history = model.fit(dataset,epochs=100)
```

```
Epoch 1/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 2s 20ms/step - loss: 48.3064 - mae: 48.8040
Epoch 2/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 11.9202 - mae: 12.4115
Epoch 3/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.6620 - mae: 10.1499
Epoch 4/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 8.4081 - mae: 8.8956
Epoch 5/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.9188 - mae: 8.4062
Epoch 6/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.6414 - mae: 8.1287
Epoch 7/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.4240 - mae: 7.9078
Epoch 8/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.2194 - mae: 7.7022
Epoch 9/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 7.0240 - mae: 7.5040
Epoch 10/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.8616 - mae: 7.3388
Epoch 11/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.7286 - mae: 7.2056
Epoch 12/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.6133 - mae: 7.0927
Epoch 13/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.5246 - mae: 7.0038
Epoch 14/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.4264 - mae: 6.9028
Epoch 15/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.3362 - mae: 6.8115
Epoch 16/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.2565 - mae: 6.7325
Epoch 17/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.1828 - mae: 6.6591
Epoch 18/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.1153 - mae: 6.5922
Epoch 19/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 6.0514 - mae: 6.5286
Epoch 20/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.9909 - mae: 6.4686
Epoch 21/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.9342 - mae: 6.4117
Epoch 22/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.8815 - mae: 6.3574
Epoch 23/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.8335 - mae: 6.3078
Epoch 24/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.7935 - mae: 6.2680
Epoch 25/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.7592 - mae: 6.2339
Epoch 26/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.7257 - mae: 6.2012
Epoch 27/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.6914 - mae: 6.1675
Epoch 28/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.6550 - mae: 6.1303
Epoch 29/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.6200 - mae: 6.0947
Epoch 30/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.5868 - mae: 6.0629
Epoch 31/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.5594 - mae: 6.0364
Epoch 32/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.5363 - mae: 6.0131
Epoch 33/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.5124 - mae: 5.9887
Epoch 34/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.4899 - mae: 5.9664
Epoch 35/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.4680 - mae: 5.9452
Epoch 36/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.4452 - mae: 5.9227
Epoch 37/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.4212 - mae: 5.8988
Epoch 38/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.3978 - mae: 5.8752
Epoch 39/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.3757 - mae: 5.8528
```

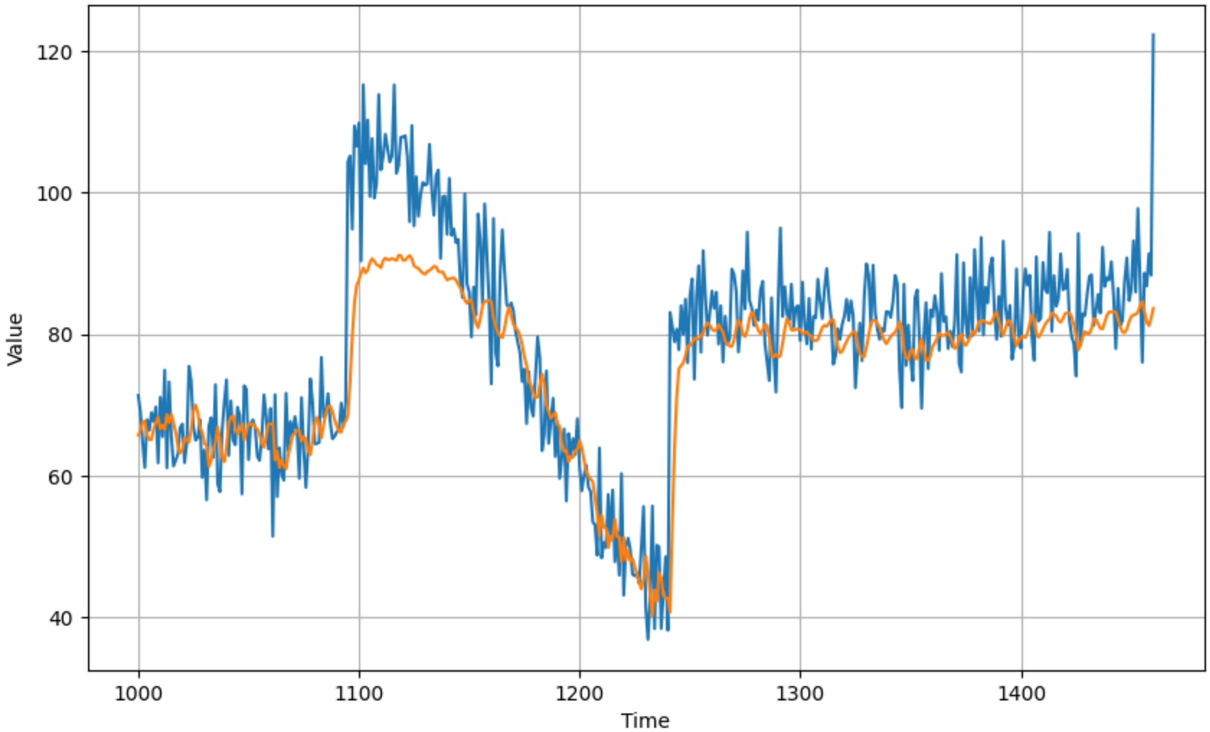
```
Epoch 40/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.3549 - mae: 5.8325
Epoch 41/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.3340 - mae: 5.8116
Epoch 42/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.3127 - mae: 5.7896
Epoch 43/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.2918 - mae: 5.7686
Epoch 44/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.2715 - mae: 5.7482
Epoch 45/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.2515 - mae: 5.7279
Epoch 46/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.2317 - mae: 5.7087
Epoch 47/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.2128 - mae: 5.6901
Epoch 48/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1973 - mae: 5.6737
Epoch 49/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1838 - mae: 5.6615
Epoch 50/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1708 - mae: 5.6488
Epoch 51/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1584 - mae: 5.6371
Epoch 52/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1491 - mae: 5.6283
Epoch 53/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1424 - mae: 5.6210
Epoch 54/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1354 - mae: 5.6151
Epoch 55/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1281 - mae: 5.6080
Epoch 56/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1211 - mae: 5.6012
Epoch 57/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1127 - mae: 5.5929
Epoch 58/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.1033 - mae: 5.5834
Epoch 59/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0937 - mae: 5.5738
Epoch 60/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0838 - mae: 5.5638
Epoch 61/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0729 - mae: 5.5528
Epoch 62/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0606 - mae: 5.5400
Epoch 63/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0482 - mae: 5.5273
Epoch 64/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0363 - mae: 5.5154
Epoch 65/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0246 - mae: 5.5038
Epoch 66/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5.0122 - mae: 5.4915
Epoch 67/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9990 - mae: 5.4783
Epoch 68/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9857 - mae: 5.4647
Epoch 69/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9725 - mae: 5.4509
Epoch 70/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9603 - mae: 5.4382
Epoch 71/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9490 - mae: 5.4264
Epoch 72/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9379 - mae: 5.4151
Epoch 73/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9270 - mae: 5.4038
Epoch 74/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9165 - mae: 5.3928
Epoch 75/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9092 - mae: 5.3854
Epoch 76/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.9029 - mae: 5.3791
Epoch 77/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8962 - mae: 5.3727
Epoch 78/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8887 - mae: 5.3655
```

```
Epoch 79/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8809 - mae: 5.3578
Epoch 80/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8732 - mae: 5.3503
Epoch 81/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8658 - mae: 5.3430
Epoch 82/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8584 - mae: 5.3356
Epoch 83/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8511 - mae: 5.3283
Epoch 84/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8438 - mae: 5.3210
Epoch 85/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8366 - mae: 5.3138
Epoch 86/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8295 - mae: 5.3067
Epoch 87/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8224 - mae: 5.2996
Epoch 88/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8153 - mae: 5.2926
Epoch 89/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8082 - mae: 5.2855
Epoch 90/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.8011 - mae: 5.2784
Epoch 91/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7941 - mae: 5.2713
Epoch 92/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7870 - mae: 5.2642
Epoch 93/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7800 - mae: 5.2572
Epoch 94/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7732 - mae: 5.2503
Epoch 95/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7666 - mae: 5.2437
Epoch 96/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7601 - mae: 5.2371
Epoch 97/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7536 - mae: 5.2304
Epoch 98/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7470 - mae: 5.2238
Epoch 99/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7404 - mae: 5.2172
Epoch 100/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4.7336 - mae: 5.2104
```

Model Prediction

Now it's time to generate the model predictions for the validation set time range. The model is a lot bigger than the ones you used before and the sequential nature of RNNs (i.e. inputs go through a series of time steps as opposed to parallel processing) can make predictions a bit slow. You can observe this when using the code you ran in the previous lab. This will take about a minute to complete.

In [14]:
```python
# Initialize a list
forecast = []

# Reduce the original series
forecast_series = series[split_time - window_size:]

# Use the model to predict data points per window size
for time in range(len(forecast_series) - window_size):
  forecast.append(model.predict(forecast_series[time:time + window_size][np.newaxis], verbose=0))

# Convert to a numpy array and drop single dimensional axes
results = np.array(forecast).squeeze()

# Plot the results
plot_series(time_valid, (x_valid, results))
```



You can optimize this step by leveraging Tensorflow models' capability to process batches. Instead of running the for-loop above which processes a single window at a time, you can pass in an entire batch of windows and let the model process that in parallel.

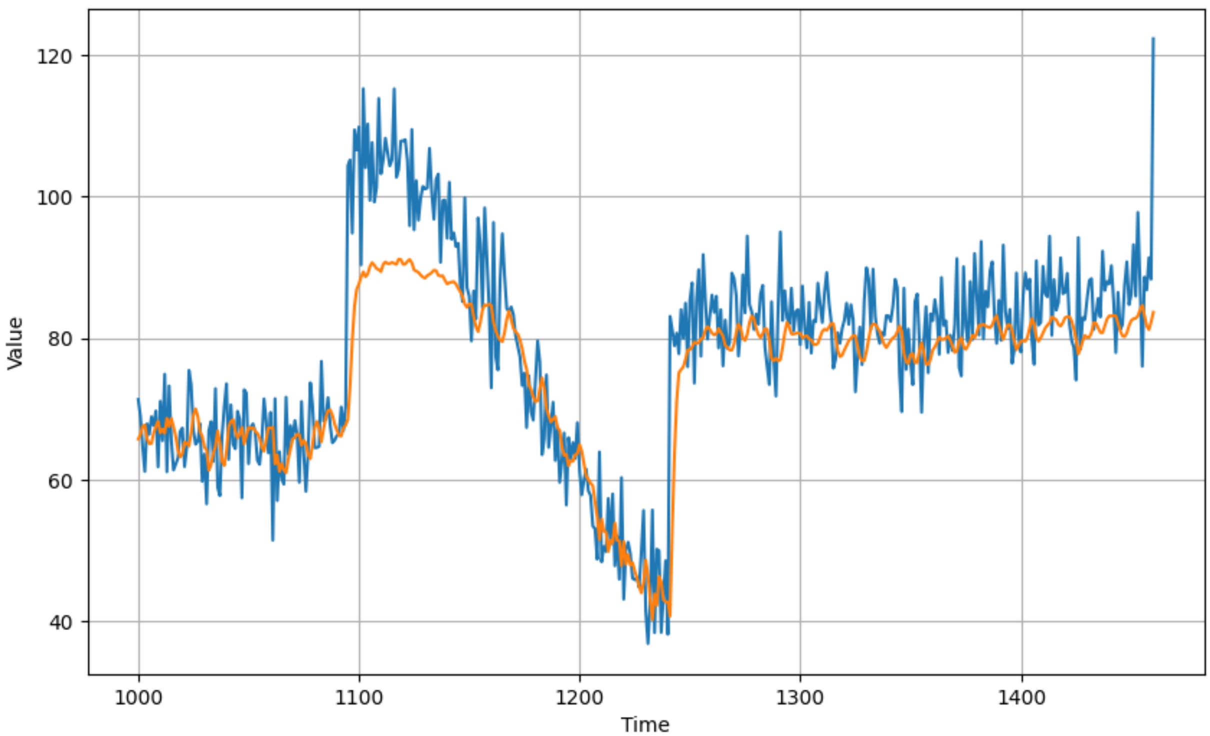
The function below does just that. You will notice that it almost mirrors the `windowed_dataset()` function but it does not shuffle the windows. That's because we want the output to be in its proper sequence so we can compare it properly to the validation set.

In [15]:
```
def model_forecast(model, series, window_size, batch_size):
    """Uses an input model to generate predictions on data windows

    Args:
      model (TF Keras Model) - model that accepts data windows
      series (array of float) - contains the values of the time series
      window_size (int) - the number of time steps to include in the window
      batch_size (int) - the batch size

    Returns:
      forecast (numpy array) - array containing predictions
    """

    # Add an axis for the feature dimension of RNN layers
    series = tf.expand_dims(series, axis=-1)

    # Generate a TF Dataset from the series values
    dataset = tf.data.Dataset.from_tensor_slices(series)

    # Window the data but only take those with the specified size
    dataset = dataset.window(window_size, shift=1, drop_remainder=True)

    # Flatten the windows by putting its elements in a single batch
    dataset = dataset.flat_map(lambda w: w.batch(window_size))

    # Create batches of windows
    dataset = dataset.batch(batch_size).prefetch(1)

    # Get predictions on the entire dataset
    forecast = model.predict(dataset, verbose=0)

    return forecast
```

You can run the function below to use the function. Notice that the predictions are generated almost instantly.

Note: You might notice that the first line slices the `series` at `split_time - window_size:-1` which is a bit different from the slower for-loop code. That is because we want the model to have its last prediction to align with the last point of the validation set (i.e. `t=1460`). You were able to do that with the slower for-loop code by specifying the for-loop's `range()`. With the more efficient function above, you *don't have that mechanism so you instead just remove the last point when slicing the `series`. If you don't, then the function will generate a prediction at `t=1461` which is outside the validation set range.*

In [16]:
```
# Only needed in this lab. Reset the model but keep the trained weights to prepare for batched inputs.
model.compile(loss=tf.keras.losses.Huber(),
              optimizer=optimizer,
              metrics=["mae"])

# Reduce the original series
forecast_series = series[split_time - window_size:-1]

# Use helper function to generate predictions
forecast = model_forecast(model, forecast_series, window_size, batch_size)

# Drop single dimensional axis
results = forecast.squeeze()

# Plot the results
plot_series(time_valid, (x_valid, results))
```



You can then compute the MSE and MAE. You can compare the results here when using other RNN architectures which you'll do in the next lab.

In [17]:
```
# Compute the MSE and MAE
print(tf.keras.metrics.mse(x_valid, results).numpy())
print(tf.keras.metrics.mae(x_valid, results).numpy())
```

```
67.760345
5.9807267
```

Wrap Up

In the next lab, you will explore a similar architecture but using LSTMs. Before doing so, run the cell below to free up resources. You might see a pop-up about restarting the kernel afterwards. You can safely ignore it and just press Ok. You can then close this lab, then go back to the classroom for the next lecture. See you there!

In [18]:
```
# Shutdown the kernel to free up resources.
# Note: You can expect a pop-up when you run this cell. You can safely ignore that and just press `Ok`.

from IPython import get_ipython

k = get_ipython().kernel
```

```python
k.do_shutdown(restart=False)
```

Out[18]:

```
{'status': 'ok', 'restart': False}
```