Ungraded Lab: Training a Single Layer Neural Network with Time Series Data

Now that you've seen statistical methods in the previous week, you will now shift to using neural networks to build your prediction models. You will start with a simple network in this notebook and move on to more complex architectures in the next weeks. By the end of this lab, you will be able to:

- build a single layer network and train it using the same synthetic data you used in the previous lab
- prepare time series data for training and evaluation
- measure the performance of your model against a validation set

Imports

You will first import the packages you will need to execute all the code in this lab. You will use:

- Tensorflow to build your model and prepare data windows
- Numpy for numerical processing
- and Matplotlib's PyPlot library for visualization

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

Utilities

You will then define some utility functions that you also saw in the previous labs. These will take care of visualizing your time series data and model predictions, as well as generating the synthetic data.

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
      label - tag for the line
      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()
```

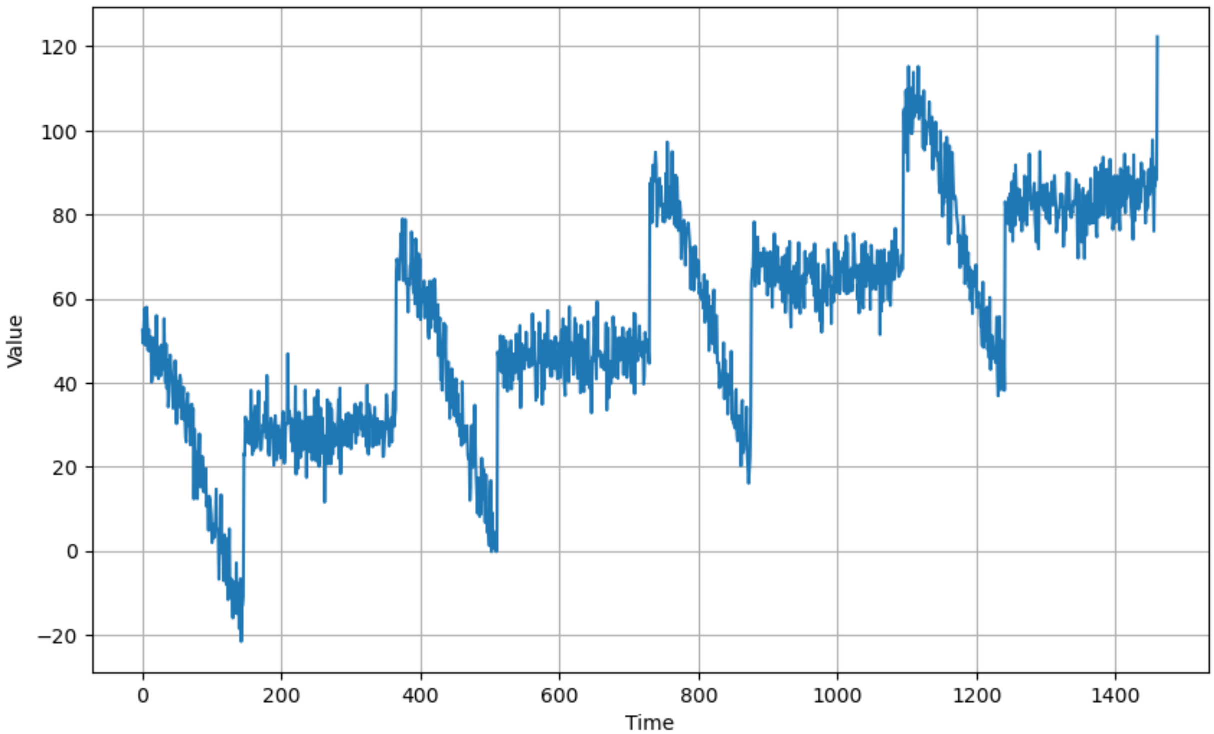
```python
def trend(time, slope=0):
    """
    Generates synthetic data that follows a straight line given a slope value.

    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
    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

The code below generates the same synthetic data you used in the previous lab. It will contain 1,461 data points that has trend, seasonality, and noise.

In [3]:
```
# 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)
```



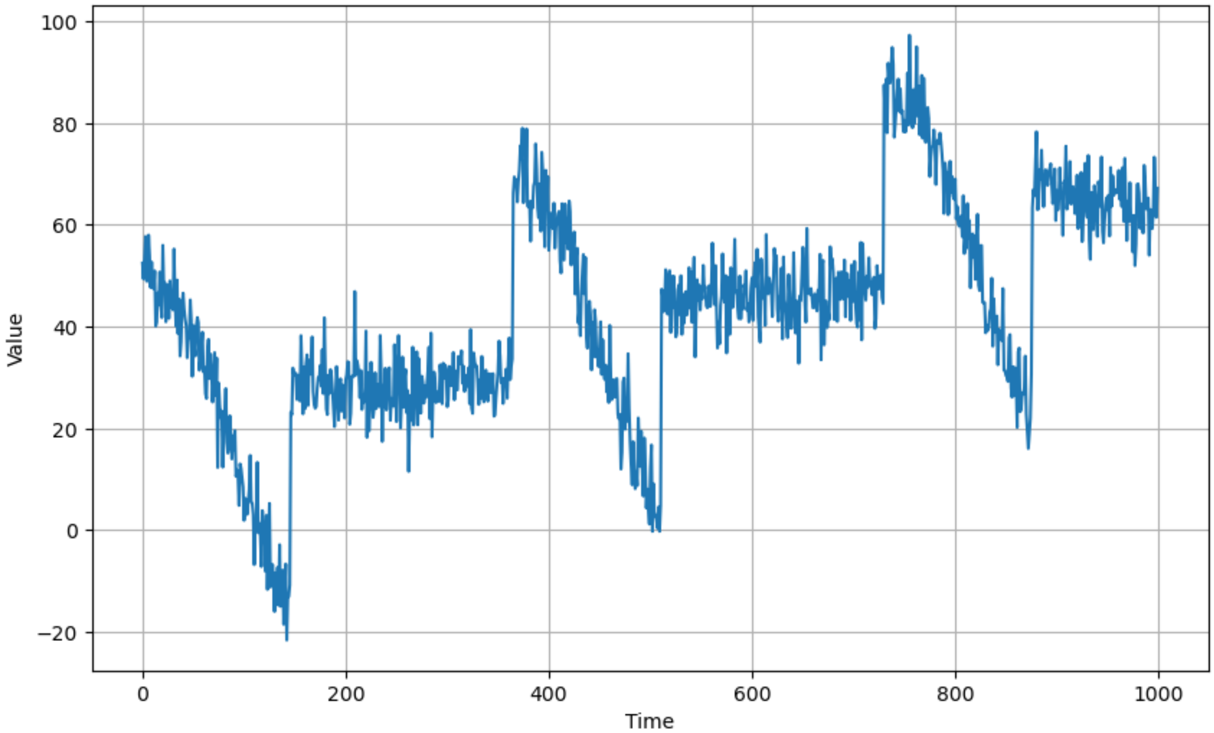
Split the Dataset

Next up, you will split the data above into training and validation sets. You will take the first 1,000 points for training while the rest is for validation.

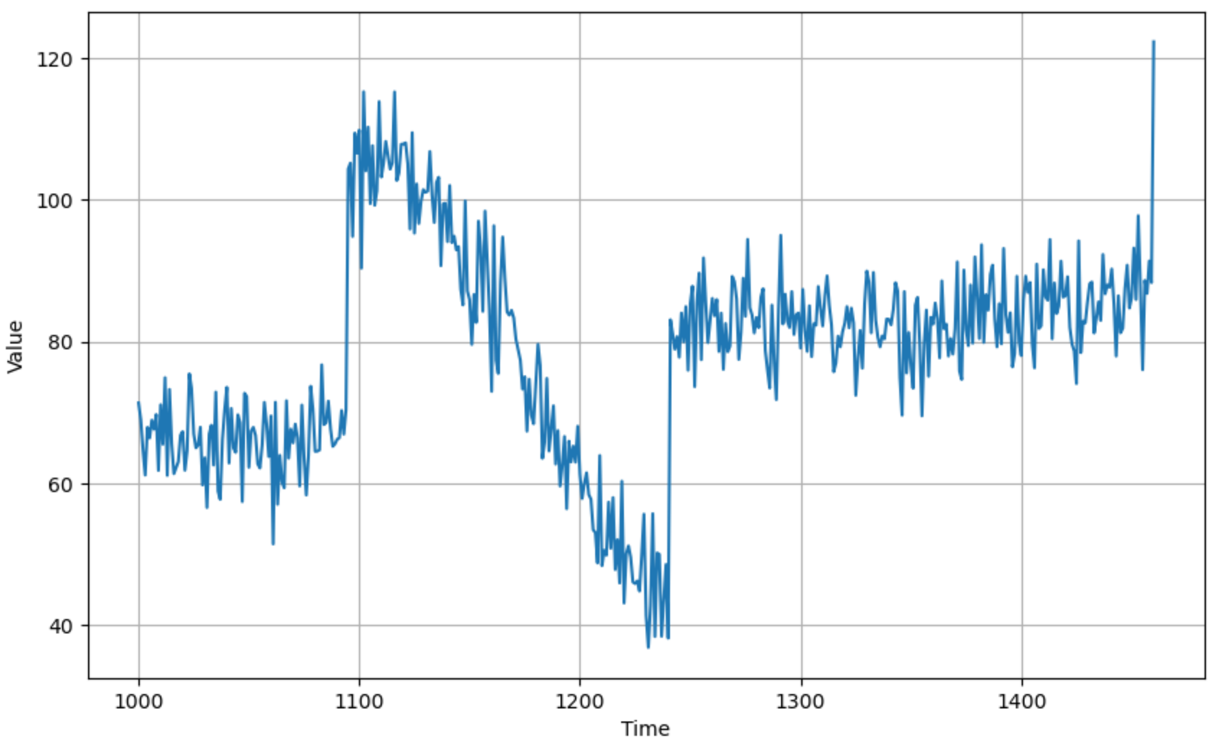
In [4]:
```
# Define the split time
split_time = 1000

# 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:]
```

You can inspect these sets visually by using the same utility function for plotting. Notice that in general, the validation set has higher values (i.e. y-axis) than those in the training set. Your model should be able to predict those values just by learning from the trend and seasonality of the training set.

In [5]:
```
# Plot the train set
plot_series(time_train, x_train)
```



In [6]:
```
# Plot the validation set
plot_series(time_valid, x_valid)
```



Prepare features and labels

You will then prepare your data windows as shown in the previous lab. It is good to declare parameters in a separate cell so you can easily tweak it later if you want.

In [7]:
```
# Parameters
window_size = 20
batch_size = 32
shuffle_buffer_size = 1000
```

The following function contains all the preprocessing steps you did in the previous lab. This makes it modular so you can easily use it in your other projects if needed.

One thing to note here is the `window_size + 1` when you call `dataset.window()`. There is a `+ 1` to indicate that you're taking the next point as the label. For example, the first 20 points will be the feature so the 21st point will be the label.

In [8]:
```
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
    """

    # 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
```

Now you can generate the dataset windows from the train set.

In [9]:
```
# Generate the dataset windows
dataset = windowed_dataset(x_train, window_size, batch_size, shuffle_buffer_size)
```

You can again inspect the output to see if the function is behaving as expected. The code below will use the `take()` method of the `tf.data.Dataset` API to grab a single batch. It will then print several properties of this batch such as the data type and shape of the elements. As expected, it should have a 2-element tuple (i.e. `(feature, label)`) and the shapes of these should align with the batch and window sizes you declared earlier which are `32` and `20` by default, respectively.

In [10]:
```
# Print properties of a single batch
for windows in dataset.take(1):
  print(f'data type: {type(windows)}')
  print(f'number of elements in the tuple: {len(windows)}')
  print(f'shape of first element: {windows[0].shape}')
  print(f'shape of second element: {windows[1].shape}')
```

```
data type: <class 'tuple'>
number of elements in the tuple: 2
shape of first element: (32, 20)
shape of second element: (32,)
```

Build and compile the model

Next, you will build the single layer neural network. This will just be a one-unit Dense layer as shown below. You will assign the layer to a variable `l0` so you can also look at the final weights later using the `get_weights()` method.

In [11]:
```
# Build the single layer neural network
l0 = tf.keras.layers.Dense(1)
model = tf.keras.models.Sequential([
    tf.keras.Input(shape=(window_size,)),
    l0
])

# Print the initial layer weights
print("Layer weights: \n {} \n".format(l0.get_weights()))

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

```
Layer weights: 
 [array([[ 0.27115786],
       [ 0.3409773 ],
       [ 0.318635  ],
       [-0.0583227 ],
       [-0.3221833 ],
       [ 0.45279056],
       [-0.10209301],
       [ 0.34542394],
       [-0.37161213],
       [ 0.05619371],
       [ 0.51307803],
       [ 0.01687419],
       [-0.21661872],
       [-0.44663724],
       [ 0.03411609],
       [-0.42872337],
       [-0.5281445 ],
       [ 0.49828583],
       [-0.28445393],
       [ 0.46844333]], dtype=float32), array([0.], dtype=float32)]
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 1)	21

Total params: 21 (84.00 B)
Trainable params: 21 (84.00 B)
Non-trainable params: 0 (0.00 B)

You will set *mean squared error (mse)* as the loss function and use *stochastic gradient descent (SGD)* to optimize the weights during training.

In [12]:
```
# Set the training parameters
model.compile(loss="mse", optimizer=tf.keras.optimizers.SGD(learning_rate=1e-6, momentum=0.9))
```

Train the Model

Now you can proceed to train your model. You will feed in the prepared data windows and run the training for 100 epochs.

In [13]:
```
# Train the model
model.fit(dataset,epochs=100)
```

```
Epoch 1/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 400.6255
Epoch 2/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 671us/step - loss: 214.1018
Epoch 3/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 735us/step - loss: 154.8285
Epoch 4/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 710us/step - loss: 123.9076
Epoch 5/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - loss: 106.7485
Epoch 6/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 659us/step - loss: 96.8574
Epoch 7/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 699us/step - loss: 90.9243
Epoch 8/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 613us/step - loss: 87.1193
Epoch 9/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 607us/step - loss: 84.4628
Epoch 10/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 682us/step - loss: 82.4269
Epoch 11/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 661us/step - loss: 80.7305
Epoch 12/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 616us/step - loss: 79.2255
Epoch 13/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - loss: 77.8350
Epoch 14/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 934us/step - loss: 76.5195
Epoch 15/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 787us/step - loss: 75.2595
Epoch 16/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 627us/step - loss: 74.0453
Epoch 17/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 730us/step - loss: 72.8723
Epoch 18/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 962us/step - loss: 71.7385
Epoch 19/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 614us/step - loss: 70.6426
Epoch 20/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 724us/step - loss: 69.5838
Epoch 21/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 986us/step - loss: 68.5614
Epoch 22/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 660us/step - loss: 67.5748
Epoch 23/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 642us/step - loss: 66.6229
Epoch 24/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 872us/step - loss: 65.7050
Epoch 25/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 743us/step - loss: 64.8200
Epoch 26/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - loss: 63.9669
Epoch 27/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 618us/step - loss: 63.1447
Epoch 28/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 703us/step - loss: 62.3523
Epoch 29/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - loss: 61.5886
Epoch 30/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 619us/step - loss: 60.8526
Epoch 31/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 644us/step - loss: 60.1433
Epoch 32/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 689us/step - loss: 59.4598
Epoch 33/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 739us/step - loss: 58.8009
Epoch 34/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 652us/step - loss: 58.1658
Epoch 35/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 726us/step - loss: 57.5536
Epoch 36/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 657us/step - loss: 56.9634
Epoch 37/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 638us/step - loss: 56.3944
Epoch 38/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 637us/step - loss: 55.8458
Epoch 39/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 728us/step - loss: 55.3168
```

```
Epoch 40/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 861us/step - loss: 54.8067
Epoch 41/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 764us/step - loss: 54.3147
Epoch 42/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 779us/step - loss: 53.8401
Epoch 43/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 953us/step - loss: 53.3824
Epoch 44/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 650us/step - loss: 52.9408
Epoch 45/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 658us/step - loss: 52.5148
Epoch 46/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 948us/step - loss: 52.1038
Epoch 47/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 642us/step - loss: 51.7072
Epoch 48/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 614us/step - loss: 51.3245
Epoch 49/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 861us/step - loss: 50.9551
Epoch 50/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 824us/step - loss: 50.5986
Epoch 51/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 617us/step - loss: 50.2545
Epoch 52/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 762us/step - loss: 49.9223
Epoch 53/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 920us/step - loss: 49.6016
Epoch 54/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 667us/step - loss: 49.2920
Epoch 55/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 625us/step - loss: 48.9930
Epoch 56/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 687us/step - loss: 48.7043
Epoch 57/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 691us/step - loss: 48.4256
Epoch 58/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 683us/step - loss: 48.1563
Epoch 59/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 775us/step - loss: 47.8963
Epoch 60/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 913us/step - loss: 47.6451
Epoch 61/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 785us/step - loss: 47.4024
Epoch 62/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 643us/step - loss: 47.1680
Epoch 63/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 759us/step - loss: 46.9415
Epoch 64/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 680us/step - loss: 46.7227
Epoch 65/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 634us/step - loss: 46.5112
Epoch 66/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 631us/step - loss: 46.3069
Epoch 67/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 669us/step - loss: 46.1094
Epoch 68/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 656us/step - loss: 45.9185
Epoch 69/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 585us/step - loss: 45.7340
Epoch 70/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 588us/step - loss: 45.5556
Epoch 71/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 45.3832
Epoch 72/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 757us/step - loss: 45.2165
Epoch 73/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 615us/step - loss: 45.0554
Epoch 74/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 746us/step - loss: 44.8995
Epoch 75/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 898us/step - loss: 44.7488
Epoch 76/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - loss: 44.6031
Epoch 77/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - loss: 44.4622
Epoch 78/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 740us/step - loss: 44.3259
```

```
Epoch 79/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 883us/step - loss: 44.1940
Epoch 80/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 722us/step - loss: 44.0665
Epoch 81/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 682us/step - loss: 43.9431
Epoch 82/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 687us/step - loss: 43.8237
Epoch 83/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 704us/step - loss: 43.7083
Epoch 84/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 660us/step - loss: 43.5965
Epoch 85/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 899us/step - loss: 43.4884
Epoch 86/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 711us/step - loss: 43.3838
Epoch 87/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 599us/step - loss: 43.2825
Epoch 88/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - loss: 43.1845
Epoch 89/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 716us/step - loss: 43.0897
Epoch 90/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - loss: 42.9978
Epoch 91/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 698us/step - loss: 42.9090
Epoch 92/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 704us/step - loss: 42.8229
Epoch 93/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 672us/step - loss: 42.7396
Epoch 94/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 621us/step - loss: 42.6590
Epoch 95/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 606us/step - loss: 42.5809
Epoch 96/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 670us/step - loss: 42.5053
Epoch 97/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 694us/step - loss: 42.4320
Epoch 98/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - loss: 42.3611
Epoch 99/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 929us/step - loss: 42.2924
Epoch 100/100
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 948us/step - loss: 42.2259
```

Out[13]:
```
<keras.src.callbacks.history.History at 0x7aaa1837d510>
```

You can see the final weights by again calling the `get_weights()` method.

In [14]:
```python
# Print the layer weights
print("Layer weights {}".format(l0.get_weights()))
```

```
Layer weights [array([[-0.04721335],
       [ 0.01698254],
       [ 0.06500397],
       [-0.0422148 ],
       [-0.0690493 ],
       [ 0.09489613],
       [-0.03763705],
       [ 0.06564226],
       [-0.11562505],
       [ 0.03175081],
       [ 0.07016681],
       [ 0.01754677],
       [-0.08439868],
       [ 0.00051356],
       [ 0.0695237 ],
       [ 0.04613458],
       [-0.04824092],
       [ 0.25989944],
       [ 0.19234033],
       [ 0.4976445 ]], dtype=float32), array([0.01246958], dtype=float32)]
```

Model Prediction

With the training finished, you can now measure the performance of your model. You can generate a model prediction by passing a batch of data windows. If you will be slicing a window from the original `series` array, you will need to add a batch dimension before passing it to the model. That can be done by indexing with the `np.newaxis` constant or using the `np.expand_dims()` method.

In [15]:
```
# Shape of the first 20 data points slice
print(f'shape of series[0:20]: {series[0:20].shape}')

# Shape after adding a batch dimension
print(f'shape of series[0:20][np.newaxis]: {series[0:20][np.newaxis].shape}')

# Shape after adding a batch dimension (alternate way)
print(f'shape of series[0:20][np.newaxis]: {np.expand_dims(series[0:20], axis=0).shape}')

# Sample model prediction
print(f'model prediction: {model.predict(series[0:20][np.newaxis])}')
```

```
shape of series[0:20]: (20,)
shape of series[0:20][np.newaxis]: (1, 20)
shape of series[0:20][np.newaxis]: (1, 20)
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
model prediction: [[43.45532]]
```

To compute the metrics, you will want to generate model predictions for your validation set. Remember that this set refers to points at index `1000` to `1460` of the entire series. You will need to code the steps to generate those from your model. The cell below demonstrates one way of doing that.

Basically, it feeds the entire series to your model 20 points at a time and append all results to a `forecast` list. It will then slice the points that corresponds to the validation set.

The slice index below is `split_time - window_size:` because the forecast list is smaller than the series by 20 points (i.e. the window size). Since the window size is 20, the first data point in the `forecast` list corresponds to the prediction for time at index `20`. You cannot make predictions at index `0` to `19` because those are smaller than the window size. Thus, when you slice with `split_time - window_size:`, you will be getting the points at the time indices that aligns with those in the validation set.

Note: You might notice that this cell takes a while to run. In the next two labs, you will see other approaches to generating predictions to make the code run faster. You might already have some ideas and feel free to try them out after completing this lab.

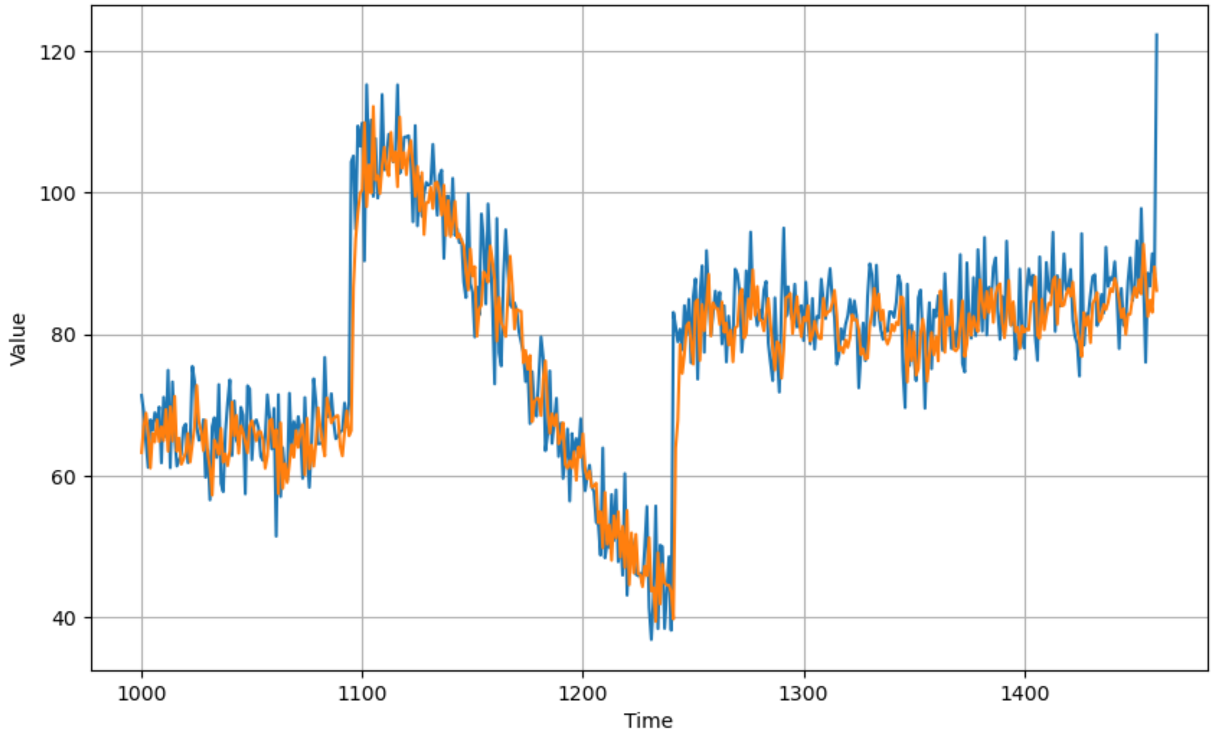
In [16]:
```
# Initialize a list
forecast = []

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

# Slice the points that are aligned with the validation set
forecast = forecast[split_time - window_size:]

# Compare number of elements in the predictions and the validation set
print(f'length of the forecast list: {len(forecast)}')
print(f'shape of the validation set: {x_valid.shape}')
```

```
length of the forecast list: 461
shape of the validation set: (461,)
```

To visualize the results, you will need to convert the predictions to a form that the `plot_series()` utility function accepts. That involves converting the list to a numpy array and dropping the single dimensional axes.

In [17]:
```
# Preview shapes after using the conversion and squeeze methods
print(f'shape after converting to numpy array: {np.array(forecast).shape}')
print(f'shape after squeezing: {np.array(forecast).squeeze().shape}')

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

# Overlay the results with the validation set
plot_series(time_valid, (x_valid, results))
```

```
shape after converting to numpy array: (461, 1, 1)
shape after squeezing: (461,)
```



You can compute the metrics by calling the same functions as before. You will get an MAE close to 5.

In [18]:
```
# Compute the metrics
print(tf.keras.metrics.mse(x_valid, results).numpy())
print(tf.keras.metrics.mae(x_valid, results).numpy())
```

```
49.041924
5.1512275
```

Wrap Up

In this lab, you were able to build and train a single layer neural network on time series data. You prepared data windows, fed them to the model, and the final predictions show comparable results with the statistical analysis you did in Week 1. In the next labs, you will try adding more layers and will also look at some optimizations you can make when training your model.