Week 3: Using RNNs to predict time series

Welcome! In the previous assignment you used a vanilla deep neural network to create forecasts for generated time series. This time you will be using Tensorflow's layers for processing sequence data such as Recurrent layers or LSTMs to see how these two approaches compare.

TIPS FOR SUCCESSFUL GRADING OF YOUR ASSIGNMENT:

- All cells are frozen except for the ones where you need to submit your solutions or when explicitly mentioned you can interact with it.
- You can add new cells to experiment but these will be omitted by the grader, so don't rely on newly created cells to host your solution code, use the provided places for this.
- You can add the comment # grade-up-to-here in any graded cell to signal the grader that it must only evaluate up to that point. This is helpful if you want to check if you are on the right track even if you are not done with the whole assignment. Be sure to remember to delete the comment afterwards!
- Avoid using global variables unless you absolutely have to. The grader tests your code in an isolated environment without running all cells from the top. As a result, global variables may be unavailable when scoring your submission. Global variables that are meant to be used will be defined in UPPERCASE.
- To submit your notebook, save it and then click on the blue submit button at the beginning of the page.

Let's get started!

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

In [2]:
```
import unittests
```

Generating the data

Let's begin by defining a bunch of helper functions to generate and plot the time series:

In [3]:
```
def plot_series(time, series, format="-", start=0, end=None):
    """Plot the series"""
    plt.plot(time[start:end], series[start:end], format)
    plt.xlabel("Time")
    plt.ylabel("Value")
    plt.grid(False)

def trend(time, slope=0):
    """A trend over time"""
    return slope * time

def seasonal_pattern(season_time):
    """Just an arbitrary pattern, you can change it if you wish"""
    return np.where(season_time < 0.1,
                    np.cos(season_time * 6 * np.pi),
                    2 / np.exp(9 * season_time))

def seasonality(time, period, amplitude=1, phase=0):
    """Repeats the same pattern at each period"""
    season_time = ((time + phase) % period) / period
    return amplitude * seasonal_pattern(season_time)

def noise(time, noise_level=1, seed=None):
    """Adds noise to the series"""
    rnd = np.random.RandomState(seed)
    return rnd.randn(len(time)) * noise_level
```

These are the same you have been using in the previous assignments, so you will be generating the same time series data. You can do that with the following function:

In [4]:
```
def generate_time_series():
```

```
    """ Creates timestamps and values of the time series """

    # The time dimension or the x-coordinate of the time series
    time = np.arange(4 * 365 + 1, dtype="float32")

    # Initial series is just a straight line with a y-intercept
    y_intercept = 10
    slope = 0.005
    series = trend(time, slope) + y_intercept

    # Adding seasonality
    amplitude = 50
    series += seasonality(time, period=365, amplitude=amplitude)

    # Adding some noise
    noise_level = 3
    series += noise(time, noise_level, seed=51)

    return time, series
```

Defining some useful global variables

Next, you will define some global variables that will be used throughout the assignment. Feel free to reference them in the upcoming exercises:

`SPLIT_TIME` : time index to split between train and validation sets

`WINDOW_SIZE` : length od the window to use for smoothing the series

`BATCH_SIZE` : batch size for training the model

`SHUFFLE_BUFFER_SIZE` : number of elements from the dataset used to sample for a new shuffle of the dataset. For more information about the use of this variable you can take a look at the docs.

A note about grading:

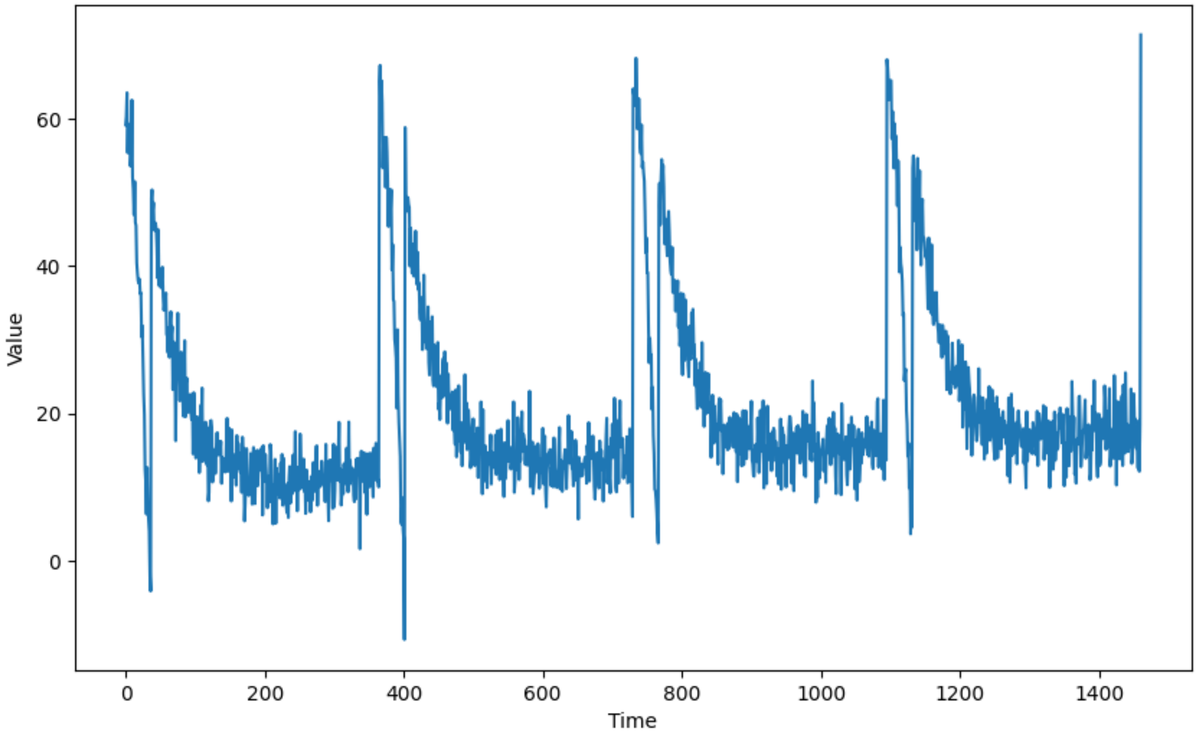
When you submit this assignment for grading these same values for these globals will be used so make sure that all your code works well with these values. After submitting and passing this assignment, you are encouraged to come back here and play with these parameters to see the impact they have in the classification process. Since this next cell is frozen, you will need to copy the contents into a new cell and run it to overwrite the values for these globals.

In [5]:
```
SPLIT_TIME = 1100
WINDOW_SIZE = 20
BATCH_SIZE = 32
SHUFFLE_BUFFER_SIZE = 1000
```

Finally, put everything together and create the times series you will use for this assignment. You will save them in the global variables `TIME` and `SERIES`.

In [6]:
```
# Create the time series
TIME, SERIES = generate_time_series()
```

In [7]:
```
# Plot the generated series
plt.figure(figsize=(10, 6))
plot_series(TIME, SERIES)
plt.show()
```



Processing the data

Since you already coded the `train_val_split` and `windowed_dataset` functions during past week's assignments, this time they are provided for you. Notice that in `windowed_dataset` an extra step is added which expands the series to have an extra dimension. This is done because you will be working with RNN-like layers which expect the dimensionality of its inputs to be 3 (including the batch dimension). In the previous weeks you used simple Dense layers which don't have this requirement.

In [8]:
```
def train_val_split(time, series):
    """ Splits time series into train and validation sets"""
    time_train = time[:SPLIT_TIME]
    series_train = series[:SPLIT_TIME]
    time_valid = time[SPLIT_TIME:]
    series_valid = series[SPLIT_TIME:]

    return time_train, series_train, time_valid, series_valid
```

In [9]:
```
def windowed_dataset(series, window_size):
    """Creates windowed dataset"""
    series = tf.expand_dims(series, axis=-1)
    dataset = tf.data.Dataset.from_tensor_slices(series)
    dataset = dataset.window(window_size + 1, shift=1, drop_remainder=True)
    dataset = dataset.flat_map(lambda window: window.batch(window_size + 1))
    dataset = dataset.shuffle(SHUFFLE_BUFFER_SIZE)
    dataset = dataset.map(lambda window: (window[:-1], window[-1]))
    dataset = dataset.batch(BATCH_SIZE).prefetch(1)
    return dataset
```

Now, run the cell below to call these two functions and generate your training dataset:

In [10]:
```
# Split the dataset
time_train, series_train, time_valid, series_valid = train_val_split(TIME, SERIES)
# Apply the transformation to the training set
dataset = windowed_dataset(series_train, WINDOW_SIZE)
```

Defining the model architecture

Exercise 1: create_uncompiled_model

Now that you have a function that will process the data before it is fed into your neural network for training, it is time to define your layer architecture.

In previous weeks or courses you defined your layers and compiled the model in the same function. However, here you will do thing a little bit different: you will first define the `create_uncompiled_model` function, which only determines your model's structure, and later on you will compile it. This way you can can reuse your model's layers for the learning rate adjusting and the actual training.

Remember that, as you saw on the lectures, there are a couple of layers you will need to add. Firstly, since LSTM and RNN layers expect three dimensions for the input (`batch_size` , `window_size` , `series_dimensionality`), and you have just a univariate time series, you will need to account for this, which can be done via the `tf.keras.Input` (this is already provided for you). Also, it is a good practice to add a layer at the end to make the output values, which are between -1 and 1 for the tanh activation function, be of the same order as the actual values of the series.

Hint:

- You should use `SimpleRNN` or `Bidirectional(LSTM)` as intermediate layers.
- The last layer of the network (before the last `Lambda`) should be a `Dense` layer.
- Fill in the `Lambda` layer at the end of the network with the correct lambda function.

In [11]:
```
# GRADED FUNCTION: create_uncompiled_model
def create_uncompiled_model():
    """Define uncompiled model

    Returns:
        tf.keras.Model: uncompiled model
    """
    ### START CODE HERE ###

    model = tf.keras.models.Sequential([
        tf.keras.Input((WINDOW_SIZE, 1)),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32, return_sequences=True)),
        tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32)),
        tf.keras.layers.Dense(1),
        tf.keras.layers.Lambda(lambda x: x * 100.0)
    ])

    ### END CODE HERE ###

    return model
```

The next cell allows you to check the number of total and trainable parameters of your model and prompts a warning in case these exceeds those of a reference solution, this serves the following 3 purposes listed in order of priority:

- Helps you prevent crashing the kernel during training.
- Helps you avoid longer-than-necessary training times.
- Provides a reasonable estimate of the size of your model. In general you will usually prefer smaller models given that they accomplish their goal successfully.

Notice that this is just informative and may be very well below the actual limit for size of the model necessary to crash the kernel. So even if you exceed this reference you are probably fine. However, **if the kernel crashes during training or it is taking a very long time and your model is larger than the reference, come back here and try to get the number of parameters closer to the reference.**

In [12]:
```
# Define your uncompiled model
uncompiled_model = create_uncompiled_model()

# Check the parameter count against a reference solution
unittests.parameter_count(uncompiled_model)
```

```
Your model has 33,601 total parameters and the reference is 35,000. You are good to go!

Your model has 33,601 trainable parameters and the reference is 35,000. You are good to go!
```

In [13]:
```
example_batch = dataset.take(1)

try:
        predictions = uncompiled_model.predict(example_batch, verbose=False)
except:
        print("Your model is not compatible with the dataset you defined earlier. Check that the loss function and last lay
else:
        print("Your current architecture is compatible with the windowed dataset! :)")
        print(f"predictions have shape: {predictions.shape}")
```

```
Your current architecture is compatible with the windowed dataset! :)
predictions have shape: (32, 1)
```

Expected output:

```
Your current architecture is compatible with the windowed dataset! :)
predictions have shape: (NUM_BATCHES, 1)
```

Where `NUM_BATCHES` is the number of batches you have set to your dataset.

In [14]:
```
# Test your code!
unittests.test_create_uncompiled_model(create_uncompiled_model)
```

```
All tests passed!
```

As a last check, you can also print a summary of your model to see what the architecture looks like. This can be useful to get a sense of how big your model is.

In [15]:
```
uncompiled_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
bidirectional (Bidirectional)	(None, 20, 64)	8,704
bidirectional_1 (Bidirectional)	(None, 64)	24,832
dense (Dense)	(None, 1)	65
lambda (Lambda)	(None, 1)	0

Total params: 33,601 (131.25 KB)
Trainable params: 33,601 (131.25 KB)
Non-trainable params: 0 (0.00 B)

Adjusting the learning rate - (Optional Exercise)

As you saw in the lectures you can leverage Tensorflow's callbacks to dinamically vary the learning rate during training. This can be helpful to get a better sense of which learning rate better accommodates to the problem at hand.

Notice that this is only changing the learning rate during the training process to give you an idea of what a reasonable learning rate is and should not be confused with selecting the best learning rate, this is known as hyperparameter optimization and it is outside the scope of this course.

For the optimizers you can try out:

- `tf.keras.optimizers.Adam`
- `tf.keras.optimizers.SGD` with a momentum of 0.9

In [16]:
```
def adjust_learning_rate(model):
    """Fit model using different learning rates

    Args:
        model (tf.keras.Model): uncompiled model

    Returns:
        tf.keras.callbacks.History: callback history
    """

    lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-6 * 10**(epoch / 20))

    ### START CODE HERE ###

    # Select your optimizer
    optimizer = tf.keras.optimizers.SGD(momentum=0.9)

    # Compile the model passing in the appropriate loss
    model.compile(loss=tf.keras.losses.Huber(),
                  optimizer=optimizer,
                  metrics=["mae"])
```

```python
    ### END CODE HERE ###

    history = model.fit(dataset, epochs=100, callbacks=[lr_schedule])

    return history
```

In [17]:
```python
# Run the training with dynamic LR
lr_history = adjust_learning_rate(uncompiled_model)
```

```
Epoch 1/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - loss: 35.9470 - mae: 36.4437 - learning_rate: 1.0000e-06
Epoch 2/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 8.0090 - mae: 8.4906 - learning_rate: 1.1220e-06
Epoch 3/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.6458 - mae: 6.1130 - learning_rate: 1.2589e-06
Epoch 4/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.8059 - mae: 5.2732 - learning_rate: 1.4125e-06
Epoch 5/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.6358 - mae: 5.1004 - learning_rate: 1.5849e-06
Epoch 6/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.6622 - mae: 5.1234 - learning_rate: 1.7783e-06
Epoch 7/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 4.5581 - mae: 5.0180 - learning_rate: 1.9953e-06
Epoch 8/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.3084 - mae: 4.7715 - learning_rate: 2.2387e-06
Epoch 9/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.9868 - mae: 4.4495 - learning_rate: 2.5119e-06
Epoch 10/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.8985 - mae: 4.3594 - learning_rate: 2.8184e-06
Epoch 11/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.5896 - mae: 4.0543 - learning_rate: 3.1623e-06
Epoch 12/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.3979 - mae: 3.8579 - learning_rate: 3.5481e-06
Epoch 13/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.7469 - mae: 4.2125 - learning_rate: 3.9811e-06
Epoch 14/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.7250 - mae: 4.1893 - learning_rate: 4.4668e-06
Epoch 15/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.4115 - mae: 3.8674 - learning_rate: 5.0119e-06
Epoch 16/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.1828 - mae: 3.6405 - learning_rate: 5.6234e-06
Epoch 17/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.3347 - mae: 3.8015 - learning_rate: 6.3096e-06
Epoch 18/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.4600 - mae: 3.9370 - learning_rate: 7.0795e-06
Epoch 19/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.1216 - mae: 3.5841 - learning_rate: 7.9433e-06
Epoch 20/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.2394 - mae: 3.6972 - learning_rate: 8.9125e-06
Epoch 21/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.5329 - mae: 3.9989 - learning_rate: 1.0000e-05
Epoch 22/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.4234 - mae: 3.8839 - learning_rate: 1.1220e-05
Epoch 23/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.0118 - mae: 3.4724 - learning_rate: 1.2589e-05
Epoch 24/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.1098 - mae: 3.5745 - learning_rate: 1.4125e-05
Epoch 25/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.2478 - mae: 3.7200 - learning_rate: 1.5849e-05
Epoch 26/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.3379 - mae: 3.8092 - learning_rate: 1.7783e-05
Epoch 27/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 2.9336 - mae: 3.3949 - learning_rate: 1.9953e-05
Epoch 28/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.1473 - mae: 3.6107 - learning_rate: 2.2387e-05
Epoch 29/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.3633 - mae: 3.8282 - learning_rate: 2.5119e-05
Epoch 30/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.1795 - mae: 3.6414 - learning_rate: 2.8184e-05
Epoch 31/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.6195 - mae: 4.0955 - learning_rate: 3.1623e-05
Epoch 32/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.9087 - mae: 4.3802 - learning_rate: 3.5481e-05
Epoch 33/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 2.9074 - mae: 3.3685 - learning_rate: 3.9811e-05
Epoch 34/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.4816 - mae: 3.9543 - learning_rate: 4.4668e-05
Epoch 35/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.3656 - mae: 3.8364 - learning_rate: 5.0119e-05
Epoch 36/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 4.3595 - mae: 4.8358 - learning_rate: 5.6234e-05
Epoch 37/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.2187 - mae: 3.6881 - learning_rate: 6.3096e-05
Epoch 38/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.2797 - mae: 3.7507 - learning_rate: 7.0795e-05
Epoch 39/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.9411 - mae: 5.4245 - learning_rate: 7.9433e-05
```

```
Epoch 40/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.8657 - mae: 4.3403 - learning_rate: 8.9125e-05
Epoch 41/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.8198 - mae: 5.2996 - learning_rate: 1.0000e-04
Epoch 42/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.3154 - mae: 4.7975 - learning_rate: 1.1220e-04
Epoch 43/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.8309 - mae: 4.3097 - learning_rate: 1.2589e-04
Epoch 44/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.6251 - mae: 5.1036 - learning_rate: 1.4125e-04
Epoch 45/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.2230 - mae: 4.7019 - learning_rate: 1.5849e-04
Epoch 46/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.5407 - mae: 5.0231 - learning_rate: 1.7783e-04
Epoch 47/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.6030 - mae: 4.0762 - learning_rate: 1.9953e-04
Epoch 48/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.6516 - mae: 5.1307 - learning_rate: 2.2387e-04
Epoch 49/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.0737 - mae: 3.5363 - learning_rate: 2.5119e-04
Epoch 50/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.1428 - mae: 5.6305 - learning_rate: 2.8184e-04
Epoch 51/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.3553 - mae: 4.8340 - learning_rate: 3.1623e-04
Epoch 52/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.1593 - mae: 4.6247 - learning_rate: 3.5481e-04
Epoch 53/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.5682 - mae: 4.0403 - learning_rate: 3.9811e-04
Epoch 54/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.3574 - mae: 4.8369 - learning_rate: 4.4668e-04
Epoch 55/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.8208 - mae: 5.3023 - learning_rate: 5.0119e-04
Epoch 56/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 4.5642 - mae: 5.0460 - learning_rate: 5.6234e-04
Epoch 57/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.2575 - mae: 3.7313 - learning_rate: 6.3096e-04
Epoch 58/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 6.5200 - mae: 7.0115 - learning_rate: 7.0795e-04
Epoch 59/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.2185 - mae: 5.7023 - learning_rate: 7.9433e-04
Epoch 60/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.4759 - mae: 5.9626 - learning_rate: 8.9125e-04
Epoch 61/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 3.6670 - mae: 4.1352 - learning_rate: 0.0010
Epoch 62/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 5.5719 - mae: 6.0501 - learning_rate: 0.0011
Epoch 63/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 3.8414 - mae: 4.3128 - learning_rate: 0.0013
Epoch 64/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 7.3462 - mae: 7.8351 - learning_rate: 0.0014
Epoch 65/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 9.4136 - mae: 9.9103 - learning_rate: 0.0016
Epoch 66/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 8.7747 - mae: 9.2695 - learning_rate: 0.0018
Epoch 67/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 11.0983 - mae: 11.5918 - learning_rate: 0.0020
Epoch 68/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 11.7176 - mae: 12.2079 - learning_rate: 0.0022
Epoch 69/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 11.9835 - mae: 12.4759 - learning_rate: 0.0025
Epoch 70/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 6.7439 - mae: 7.2271 - learning_rate: 0.0028
Epoch 71/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 25.3776 - mae: 25.8694 - learning_rate: 0.0032
Epoch 72/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 41.9443 - mae: 42.4413 - learning_rate: 0.0035
Epoch 73/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 45.2556 - mae: 45.7529 - learning_rate: 0.0040
Epoch 74/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 44.9824 - mae: 45.4799 - learning_rate: 0.0045
Epoch 75/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 46.7531 - mae: 47.2528 - learning_rate: 0.0050
Epoch 76/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 20.2534 - mae: 20.7450 - learning_rate: 0.0056
Epoch 77/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 44.9601 - mae: 45.4590 - learning_rate: 0.0063
Epoch 78/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 33.8747 - mae: 34.3718 - learning_rate: 0.0071
```

```
Epoch 79/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 40.9307 - mae: 41.4307 - learning_rate: 0.0079
Epoch 80/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 46.2117 - mae: 46.7116 - learning_rate: 0.0089
Epoch 81/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 48.1855 - mae: 48.6832 - learning_rate: 0.0100
Epoch 82/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 70.5921 - mae: 71.0907 - learning_rate: 0.0112
Epoch 83/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 213.2199 - mae: 213.7194 - learning_rate: 0.0126
Epoch 84/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 183.0053 - mae: 183.5050 - learning_rate: 0.0141
Epoch 85/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 334.4997 - mae: 334.9997 - learning_rate: 0.0158
Epoch 86/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 193.6646 - mae: 194.1644 - learning_rate: 0.0178
Epoch 87/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 111.1917 - mae: 111.6912 - learning_rate: 0.0200
Epoch 88/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 187.9846 - mae: 188.4841 - learning_rate: 0.0224
Epoch 89/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 243.8499 - mae: 244.3499 - learning_rate: 0.0251
Epoch 90/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 622.0450 - mae: 622.5450 - learning_rate: 0.0282
Epoch 91/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 696.1707 - mae: 696.6707 - learning_rate: 0.0316
Epoch 92/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 388.2701 - mae: 388.7701 - learning_rate: 0.0355
Epoch 93/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 368.2881 - mae: 368.7881 - learning_rate: 0.0398
Epoch 94/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 399.5392 - mae: 400.0392 - learning_rate: 0.0447
Epoch 95/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 565.9926 - mae: 566.4926 - learning_rate: 0.0501
Epoch 96/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 467.1892 - mae: 467.6892 - learning_rate: 0.0562
Epoch 97/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 797.5815 - mae: 798.0805 - learning_rate: 0.0631
Epoch 98/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 1219.1764 - mae: 1219.6764 - learning_rate: 0.0708
Epoch 99/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 1243.3766 - mae: 1243.8766 - learning_rate: 0.0794
Epoch 100/100
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 951.3099 - mae: 951.8099 - learning_rate: 0.0891
```

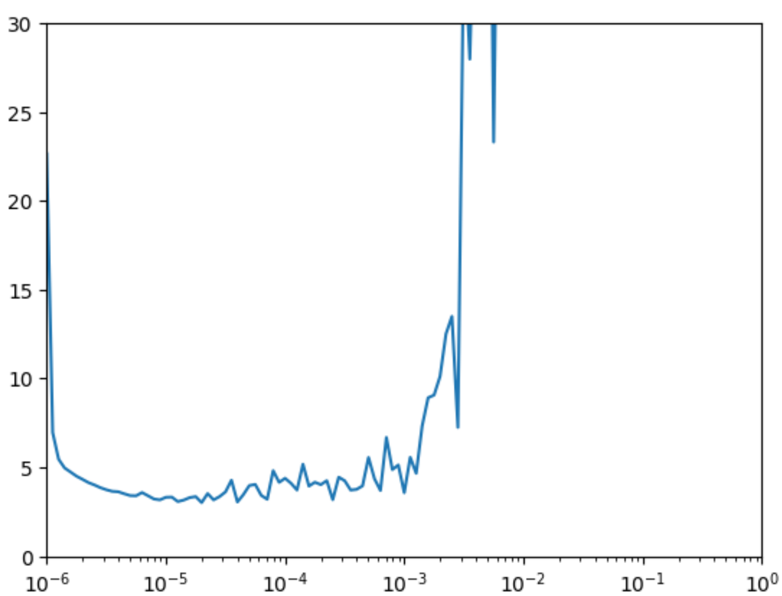
Plot the achieved loss for each learning rate value, this way you can select an appropriate learning rate for your training.

In [18]:

```python
# Plot the loss for every LR
plt.semilogx(lr_history.history["learning_rate"], lr_history.history["loss"])
plt.axis([1e-6, 1, 0, 30])
```

Out[18]:

```
(1e-06, 1.0, 0.0, 30.0)
```



Based on this plot, which learning rate would you choose? You will get to use it on the next exercise.

Compiling the model

Exercise 2: create_model

Now it is time to do the actual training that will be used to forecast the time series. For this complete the `create_model` function below.

Notice that you are reusing the architecture you defined in the `create_uncompiled_model` earlier. Now you only need to compile this model using the appropriate loss, optimizer (and learning rate). If you completed the previous optional exercise, you should have a pretty good idea of which combinations might work better.

Hint:

- The training should be really quick so if you notice that each epoch is taking more than a few seconds, consider trying a different architecture.
- If after the first epoch you get an output like this: `loss: nan - mae: nan` it is very likely that your network is suffering from exploding gradients. This is a common problem if you used `SGD` as optimizer and set a learning rate that is too high. **If you encounter this problem consider lowering the learning rate or using Adam with the default learning rate.**

In [32]:
```
# GRADED FUNCTION: create_model
def create_model():
    """Creates and compiles the model

    Returns:
        tf.keras.Model: compiled model
    """
    model = create_uncompiled_model()

    ### START CODE HERE ###

    model.compile(loss=tf.keras.losses.MeanSquaredError(),
                  optimizer=tf.keras.optimizers.Adam(learning_rate=1e-5),
                  metrics=["mae"])

    ### END CODE HERE ###

    return model
```

In [33]:
```
# Create an instance of the model
model = create_model()
```

In [34]:
```
# Test your code!
unittests.test_create_model(create_model)
```

```
All tests passed!
```

Now go ahead and train your model:

In [35]:
```
# Train it
history = model.fit(dataset, epochs=50)
```

```
Epoch 1/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - loss: 809.2649 - mae: 26.3078
Epoch 2/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 440.9386 - mae: 19.5112
Epoch 3/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 283.1550 - mae: 15.1386
Epoch 4/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 219.1285 - mae: 12.1589
Epoch 5/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 192.5452 - mae: 10.6269
Epoch 6/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 192.8171 - mae: 10.0682
Epoch 7/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 177.3301 - mae: 9.4482
Epoch 8/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 163.2354 - mae: 8.8087
Epoch 9/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 162.2510 - mae: 8.7540
Epoch 10/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 143.2909 - mae: 8.0310
Epoch 11/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 124.4082 - mae: 7.3251
Epoch 12/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 126.9431 - mae: 7.2391
Epoch 13/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 126.2300 - mae: 7.1646
Epoch 14/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 111.4987 - mae: 6.6535
Epoch 15/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 112.1032 - mae: 6.6154
Epoch 16/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 87.1638 - mae: 5.7610
Epoch 17/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 96.4278 - mae: 5.9713
Epoch 18/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 91.9683 - mae: 5.5822
Epoch 19/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 91.4548 - mae: 5.5301
Epoch 20/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 92.1121 - mae: 5.6075
Epoch 21/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 85.5951 - mae: 5.3521
Epoch 22/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 74.0902 - mae: 5.0894
Epoch 23/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 76.9980 - mae: 5.2833
Epoch 24/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 77.9492 - mae: 5.1589
Epoch 25/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 74.1703 - mae: 4.9698
Epoch 26/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 67.4111 - mae: 4.7684
Epoch 27/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 71.1877 - mae: 4.9211
Epoch 28/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 82.6750 - mae: 5.3075
Epoch 29/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 58.1885 - mae: 4.5730
Epoch 30/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 62.2923 - mae: 4.5445
Epoch 31/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 60.7535 - mae: 4.5310
Epoch 32/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 60.5052 - mae: 4.5313
Epoch 33/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 54.3105 - mae: 4.2232
Epoch 34/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 54.4676 - mae: 4.4949
Epoch 35/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 56.8973 - mae: 4.4939
Epoch 36/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 59.4638 - mae: 4.3112
Epoch 37/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 61.2833 - mae: 4.5037
Epoch 38/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 57.2288 - mae: 4.3665
Epoch 39/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 58.1055 - mae: 4.2879
```

```
Epoch 40/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 45.2223 - mae: 4.1090
Epoch 41/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 52.9028 - mae: 4.3574
Epoch 42/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 53.4874 - mae: 4.2409
Epoch 43/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 56.9006 - mae: 4.3447
Epoch 44/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 36.1171 - mae: 3.5754
Epoch 45/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 44.3604 - mae: 4.0212
Epoch 46/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 50.6242 - mae: 4.0693
Epoch 47/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 39.4033 - mae: 3.6703
Epoch 48/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 38.2569 - mae: 3.8350
Epoch 49/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 39.3252 - mae: 3.8269
Epoch 50/50
34/34 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 36.8788 - mae: 3.7787
```

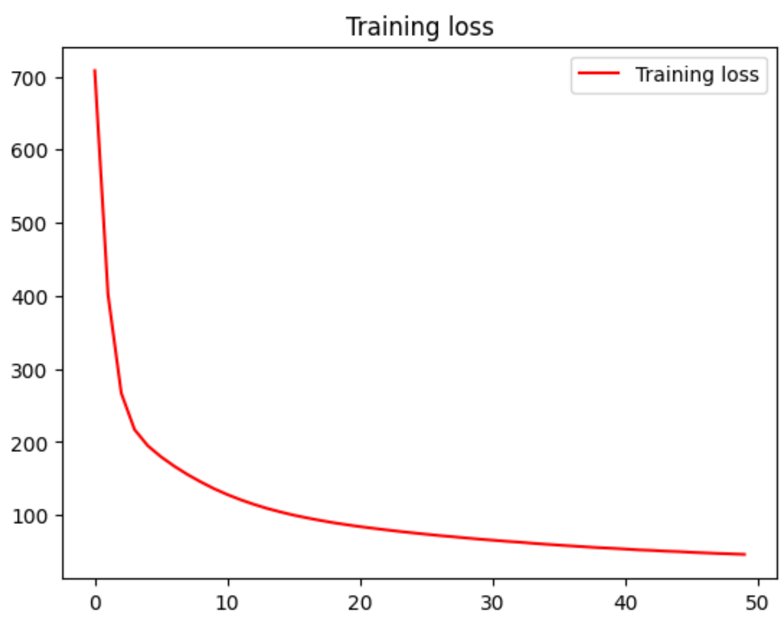
Now go ahead and plot the training loss so you can monitor the learning process.

In [36]:
```
# Plot the training loss for each epoch

loss = history.history['loss']

epochs = range(len(loss))

plt.plot(epochs, loss, 'r', label='Training loss')
plt.title('Training loss')
plt.legend(loc=0)
plt.show()
```



Evaluating the forecast

Now it is time to evaluate the performance of the forecast. For this you can use the `compute_metrics` function that you coded in a previous assignment:

In [37]:
```
def compute_metrics(true_series, forecast):
    """Computes MSE and MAE metrics for the forecast"""
    mse = tf.keras.losses.MSE(true_series, forecast)
    mae = tf.keras.losses.MAE(true_series, forecast)
    return mse, mae
```

At this point you have trained the model that will perform the forecast, but you still need to compute the actual forecast. For this, you will use the `generate_forecast` function. This function, which is the same you used on previous assignments, generates the next value given a set of the previous `window_size` points for every point in the validation set.

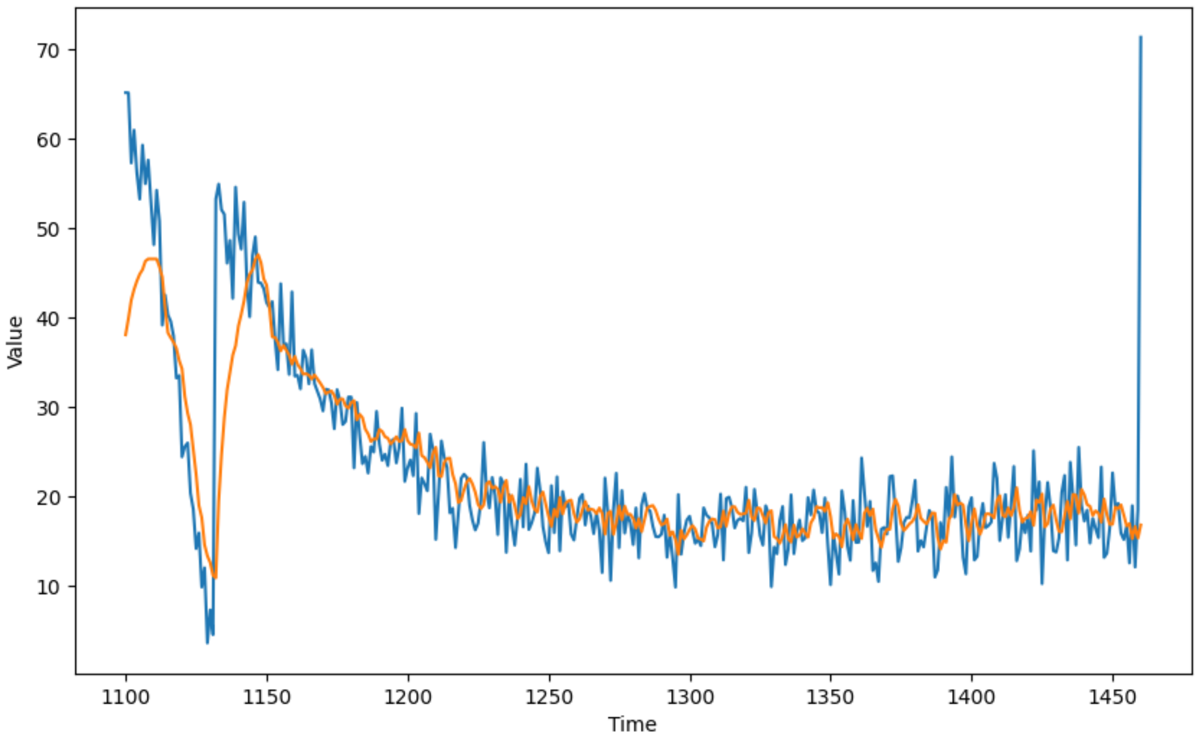
In [38]:
```
def generate_forecast(model, series, window_size):
    """Generates a forecast using your trained model"""
    forecast = []
    for time in range(SPLIT_TIME, len(series)):
        pred = model.predict(series[time-window_size:time][np.newaxis])
        forecast.append(pred[0][0])
    return forecast
```

Now, run the cells below to generate and plot the forecast series:

In [ ]:
```
# Save the forecast
rnn_forecast = generate_forecast(model, SERIES, WINDOW_SIZE)
```

In [40]:
```
# Plot your forecast
plt.figure(figsize=(10, 6))

plot_series(time_valid, series_valid)
plot_series(time_valid, rnn_forecast)
```



Expected Output:

A series similar to this one:

No description has been provided for this image

Now use the `compute_metrics function` to find the MSE and MAE of your forecast.

In [41]:
```
mse, mae = compute_metrics(series_valid, rnn_forecast)

print(f"mse: {mse:.2f}, mae: {mae:.2f} for forecast")
```

```
mse: 43.20, mae: 3.91 for forecast
```

You will be graded based on your model performance. To pass this assignment your forecast should achieve an MAE of 4.5 or less.

- If your forecast didn't achieve this threshold try re-training your model with a different architecture (you will need to re-run both `create_uncompiled_model` and `create_model` functions) or tweaking the optimizer's parameters.
- If your forecast did achieve this threshold run the following cell to save your achieved MAE for the forecast, which will be used for grading. After doing so, submit your assignment for grading.

In [42]:
```
# Save your mae in a pickle file
with open('forecast_mae.pkl', 'wb') as f:
    pickle.dump(mae.numpy(), f)
```

Congratulations on finishing this week's assignment!

You have successfully implemented a neural network capable of forecasting time series leveraging Tensorflow's layers for sequence modelling such as `RNNs` and `LSTMs` ! **This resulted in a forecast that matches (or even surpasses) the one from last week while training for half of the epochs.**

Keep it up!