

Week 4: Using real world data

Welcome! So far you have worked exclusively with generated data. This time you will be using the [Daily Minimum Temperatures in Melbourne](#) dataset which contains data of the daily minimum temperatures recorded in Melbourne from 1981 to 1990. In addition to be using Tensorflow's layers for processing sequence data such as Recurrent layers or LSTMs you will also use Convolutional layers to improve the model's performance.

- 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 csv
import pickle
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

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

Begin by looking at the structure of the csv that contains the data:

```
In [3]: DATA_PATH = './data/daily-min-temperatures.csv'

with open(DATA_PATH, 'r') as csvfile:
    print(f"Header looks like this:\n\n{csvfile.readline()}")
    print(f"First data point looks like this:\n\n{csvfile.readline()}")
    print(f"Second data point looks like this:\n\n{csvfile.readline()}")
```

Header looks like this:

```
"Date", "Temp"
```

First data point looks like this:

```
"1981-01-01", 20.7
```

Second data point looks like this:

```
"1981-01-02", 17.9
```

As you can see, each data point is composed of the date and the recorded minimum temperature for that date.

In the first exercise you will code a function to read the data from the csv but for now run the next cell to load a helper function to plot the time series.

```
In [4]: 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(True)
```

Parsing the raw data

Exercise 1: parse_data_from_file

Now you need to read the data from the csv file. To do so, complete the `parse_data_from_file` function.

A couple of things to note:

- You are encouraged to use the function `np.loadtxt` to load the data. Make sure to check out the documentation to learn about useful parameters.
- The `times` list should contain every timestep (starting at zero), which is just a sequence of ordered numbers with the same length as the `temperatures` list.
- The values of the `temperatures` should be of `float` type. Make sure to select the correct column to read with `np.loadtxt`.

```
In [9]: # GRADED FUNCTION: parse_data
def parse_data_from_file(filename):
    """Parse data from csv file

    Args:
        filename (str): complete path to file (path + filename)

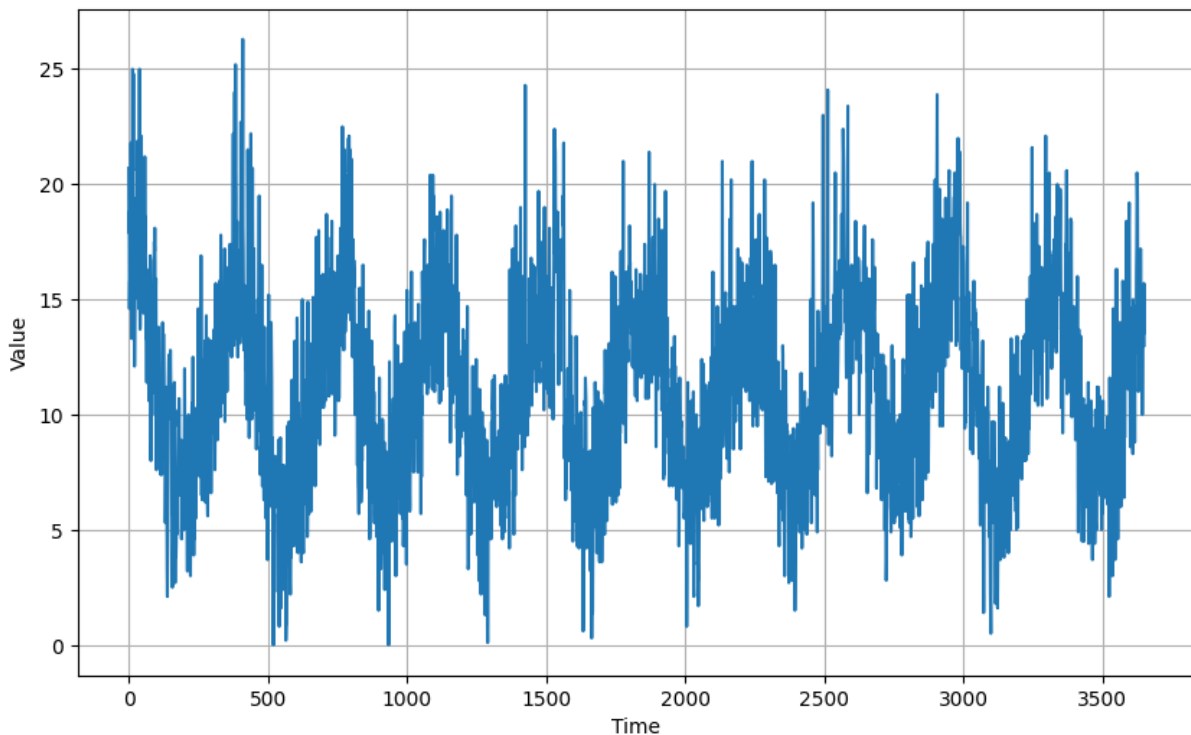
    Returns:
        (np.ndarray, np.ndarray): arrays of timestamps and values of the time series
    """
    ### START CODE HERE
    # Load the temperatures using np.loadtxt. Remember you want to skip the first
    # row, since it's headers. Make sure to use the correct column of the csv file.
    temperatures = np.loadtxt(filename, delimiter=',', dtype=float, skiprows=1, usecols=1)
    times = np.arange(len(temperatures)) # Create the time steps.
    ### END CODE HERE

    return times, temperatures
```


Now, use this function to create the timestamps, `TIME`, and the time series, `SERIES`

```
In [10]: TIME, SERIES = parse_data_from_file(DATA_PATH)
```

```
In [11]: # Plot the series!
plt.figure(figsize=(10, 6))
plot_series(TIME, SERIES)
```



Expected Output:

 No description has been provided for this image

```
In [12]: # Test your code!
unittests.test_parse_data_from_file(parse_data_from_file)

All tests passed!
```

Defining some useful global variables

Next, you will define some global variables that will come 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 of 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.

The next cell will use your function to compute the `times` and `temperatures` and will save these as numpy arrays within the `G` dataclass. This cell will also plot the time series:

```
In [13]: # Save all global variables
SPLIT_TIME = 2500
WINDOW_SIZE = 64
BATCH_SIZE = 256
SHUFFLE_BUFFER_SIZE = 1000
```

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 like in week 3, the `windowed_dataset` function has an extra step, which expands the series to have an extra dimension. This is done because you will be working with Conv layers which expect the dimensionality of its inputs to be 3 (including the batch dimension).

```
In [14]: def train_val_split(time, series):
        """ Splits time series into train and validations 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 [15]: # Split the dataset
time_train, series_train, time_valid, series_valid = train_val_split(TIME, SERIES)
```

```
In [16]: 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
```

```
In [17]: # Apply the transformation to the training set
train_dataset = windowed_dataset(series_train, window_size=WINDOW_SIZE)
```

Defining the model architecture

Exercise 2: 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 model architecture. Just as in last week's assignment you will do the layer definition and compilation in two separate steps. Begin by completing the `create_uncompiled_model` function below.

This is done so you can reuse your model's layers for the learning rate adjusting and the actual training.

Hint:

- Remember that the original dataset was expanded, so account for this when setting the shape of the `tf.keras.Input`
- No `Lambda` layers are required
- Use a combination of `Conv1D` and `LSTM` layers, followed by `Dense`.

```
In [18]: # 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(shape=(WINDOW_SIZE,1)), # Set the correct input shape for the model
        tf.keras.layers.Conv1D(filters=32, kernel_size=5,
```

```

        strides=1, padding='causal',
        activation='relu'),
tf.keras.layers.LSTM(units=64, return_sequences=True),
tf.keras.layers.LSTM(units=60),
tf.keras.layers.Dense(units=30, activation='relu'),
tf.keras.layers.Dense(units=10, activation='relu'),
tf.keras.layers.Dense(units=1)
])

### 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 [19]: # Get your uncompiled model
uncompiled_model = create_uncompiled_model()

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

```

Your model has 57,175 total parameters and the reference is 60,000. You are good to go!

Your model has 57,175 trainable parameters and the reference is 60,000. You are good to go!

```

In [20]: example_batch = train_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 layer are correct.")
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: (256, 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 [21]: # Test your code!
unittests.test_create_uncompiled_model(create_uncompiled_model)

All tests passed!

```

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 [22]: uncompiled_model.summary()

Model: "sequential"

```

Layer (type)	Output Shape	Param #
conv1d (Conv1D)	(None, 64, 32)	192
lstm (LSTM)	(None, 64, 64)	24,832
lstm_1 (LSTM)	(None, 60)	30,000
dense (Dense)	(None, 30)	1,830
dense_1 (Dense)	(None, 10)	310
dense_2 (Dense)	(None, 1)	11

Total params: 57,175 (223.34 KB)

Trainable params: 57,175 (223.34 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 dynamically 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. This is the same function you had on Week 3 Assignment, feel free to reuse it.

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 [23]: def adjust_learning_rate(dataset):
          """Fit model using different learning rates

          Args:
              dataset (tf.data.Dataset): train dataset

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

          model = create_uncompiled_model()

          lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-5 * 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"])

          ### END CODE HERE ###

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

          return history
```

```
In [24]: # Run the training with dynamic LR
          lr_history = adjust_learning_rate(train_dataset)
```

Epoch 1/100
10/10 3s 19ms/step - loss: 10.1302 - mae: 10.6294 - learning_rate: 1.0000e-05
Epoch 2/100
10/10 0s 20ms/step - loss: 10.1546 - mae: 10.6535 - learning_rate: 1.1220e-05
Epoch 3/100
10/10 0s 19ms/step - loss: 10.0940 - mae: 10.5930 - learning_rate: 1.2589e-05
Epoch 4/100
10/10 0s 19ms/step - loss: 10.0379 - mae: 10.5363 - learning_rate: 1.4125e-05
Epoch 5/100
10/10 0s 21ms/step - loss: 10.0672 - mae: 10.5662 - learning_rate: 1.5849e-05
Epoch 6/100
10/10 0s 19ms/step - loss: 10.0845 - mae: 10.5836 - learning_rate: 1.7783e-05
Epoch 7/100
10/10 0s 19ms/step - loss: 10.1591 - mae: 10.6580 - learning_rate: 1.9953e-05
Epoch 8/100
10/10 0s 22ms/step - loss: 10.0573 - mae: 10.5560 - learning_rate: 2.2387e-05
Epoch 9/100
10/10 0s 19ms/step - loss: 9.9667 - mae: 10.4654 - learning_rate: 2.5119e-05
Epoch 10/100
10/10 0s 18ms/step - loss: 9.9156 - mae: 10.4147 - learning_rate: 2.8184e-05
Epoch 11/100
10/10 0s 22ms/step - loss: 9.8435 - mae: 10.3422 - learning_rate: 3.1623e-05
Epoch 12/100
10/10 0s 21ms/step - loss: 9.7723 - mae: 10.2714 - learning_rate: 3.5481e-05
Epoch 13/100
10/10 0s 19ms/step - loss: 9.6435 - mae: 10.1422 - learning_rate: 3.9811e-05
Epoch 14/100
10/10 0s 17ms/step - loss: 9.5697 - mae: 10.0679 - learning_rate: 4.4668e-05
Epoch 15/100
10/10 0s 21ms/step - loss: 9.5940 - mae: 10.0932 - learning_rate: 5.0119e-05
Epoch 16/100
10/10 0s 19ms/step - loss: 9.4438 - mae: 9.9414 - learning_rate: 5.6234e-05
Epoch 17/100
10/10 0s 17ms/step - loss: 9.2925 - mae: 9.7907 - learning_rate: 6.3096e-05
Epoch 18/100
10/10 0s 19ms/step - loss: 9.2167 - mae: 9.7149 - learning_rate: 7.0795e-05
Epoch 19/100
10/10 0s 18ms/step - loss: 9.0433 - mae: 9.5413 - learning_rate: 7.9433e-05
Epoch 20/100
10/10 0s 21ms/step - loss: 8.8547 - mae: 9.3532 - learning_rate: 8.9125e-05
Epoch 21/100
10/10 0s 19ms/step - loss: 8.6991 - mae: 9.1967 - learning_rate: 1.0000e-04
Epoch 22/100
10/10 0s 18ms/step - loss: 8.4674 - mae: 8.9660 - learning_rate: 1.1220e-04
Epoch 23/100
10/10 0s 18ms/step - loss: 8.2190 - mae: 8.7164 - learning_rate: 1.2589e-04
Epoch 24/100
10/10 0s 19ms/step - loss: 7.8148 - mae: 8.3107 - learning_rate: 1.4125e-04
Epoch 25/100
10/10 0s 22ms/step - loss: 7.3042 - mae: 7.7990 - learning_rate: 1.5849e-04
Epoch 26/100
10/10 0s 19ms/step - loss: 6.6710 - mae: 7.1649 - learning_rate: 1.7783e-04
Epoch 27/100
10/10 0s 22ms/step - loss: 5.8217 - mae: 6.3096 - learning_rate: 1.9953e-04
Epoch 28/100
10/10 0s 19ms/step - loss: 4.8014 - mae: 5.2853 - learning_rate: 2.2387e-04
Epoch 29/100
10/10 0s 17ms/step - loss: 3.7111 - mae: 4.1834 - learning_rate: 2.5119e-04
Epoch 30/100
10/10 0s 17ms/step - loss: 2.8747 - mae: 3.3405 - learning_rate: 2.8184e-04
Epoch 31/100
10/10 0s 19ms/step - loss: 2.6032 - mae: 3.0694 - learning_rate: 3.1623e-04
Epoch 32/100
10/10 0s 20ms/step - loss: 2.5431 - mae: 3.0080 - learning_rate: 3.5481e-04
Epoch 33/100
10/10 0s 19ms/step - loss: 2.4821 - mae: 2.9439 - learning_rate: 3.9811e-04
Epoch 34/100
10/10 0s 18ms/step - loss: 2.3493 - mae: 2.8099 - learning_rate: 4.4668e-04
Epoch 35/100
10/10 0s 18ms/step - loss: 2.1931 - mae: 2.6514 - learning_rate: 5.0119e-04
Epoch 36/100
10/10 0s 18ms/step - loss: 2.2565 - mae: 2.7150 - learning_rate: 5.6234e-04
Epoch 37/100
10/10 0s 19ms/step - loss: 2.0407 - mae: 2.5006 - learning_rate: 6.3096e-04
Epoch 38/100
10/10 0s 20ms/step - loss: 1.9551 - mae: 2.4061 - learning_rate: 7.0795e-04
Epoch 39/100
10/10 0s 18ms/step - loss: 1.8886 - mae: 2.3409 - learning_rate: 7.9433e-04

Epoch 40/100
10/10 — 0s 19ms/step - loss: 1.9269 - mae: 2.3785 - learning_rate: 8.9125e-04
Epoch 41/100
10/10 — 0s 22ms/step - loss: 1.8256 - mae: 2.2759 - learning_rate: 0.0010
Epoch 42/100
10/10 — 0s 18ms/step - loss: 1.8575 - mae: 2.3036 - learning_rate: 0.0011
Epoch 43/100
10/10 — 0s 19ms/step - loss: 1.8791 - mae: 2.3324 - learning_rate: 0.0013
Epoch 44/100
10/10 — 0s 21ms/step - loss: 1.8490 - mae: 2.3004 - learning_rate: 0.0014
Epoch 45/100
10/10 — 0s 19ms/step - loss: 1.7901 - mae: 2.2409 - learning_rate: 0.0016
Epoch 46/100
10/10 — 0s 19ms/step - loss: 1.7881 - mae: 2.2386 - learning_rate: 0.0018
Epoch 47/100
10/10 — 0s 18ms/step - loss: 1.8989 - mae: 2.3532 - learning_rate: 0.0020
Epoch 48/100
10/10 — 0s 21ms/step - loss: 1.7804 - mae: 2.2292 - learning_rate: 0.0022
Epoch 49/100
10/10 — 0s 18ms/step - loss: 1.8087 - mae: 2.2595 - learning_rate: 0.0025
Epoch 50/100
10/10 — 0s 19ms/step - loss: 1.9195 - mae: 2.3712 - learning_rate: 0.0028
Epoch 51/100
10/10 — 0s 18ms/step - loss: 1.7254 - mae: 2.1771 - learning_rate: 0.0032
Epoch 52/100
10/10 — 0s 20ms/step - loss: 1.6724 - mae: 2.1186 - learning_rate: 0.0035
Epoch 53/100
10/10 — 0s 18ms/step - loss: 1.7820 - mae: 2.2323 - learning_rate: 0.0040
Epoch 54/100
10/10 — 0s 19ms/step - loss: 2.0074 - mae: 2.4622 - learning_rate: 0.0045
Epoch 55/100
10/10 — 0s 21ms/step - loss: 2.0330 - mae: 2.4892 - learning_rate: 0.0050
Epoch 56/100
10/10 — 0s 19ms/step - loss: 1.6611 - mae: 2.1099 - learning_rate: 0.0056
Epoch 57/100
10/10 — 0s 19ms/step - loss: 1.8006 - mae: 2.2522 - learning_rate: 0.0063
Epoch 58/100
10/10 — 0s 21ms/step - loss: 1.7649 - mae: 2.2156 - learning_rate: 0.0071
Epoch 59/100
10/10 — 0s 19ms/step - loss: 1.8056 - mae: 2.2600 - learning_rate: 0.0079
Epoch 60/100
10/10 — 0s 18ms/step - loss: 1.8077 - mae: 2.2615 - learning_rate: 0.0089
Epoch 61/100
10/10 — 0s 20ms/step - loss: 1.8212 - mae: 2.2773 - learning_rate: 0.0100
Epoch 62/100
10/10 — 0s 19ms/step - loss: 1.6453 - mae: 2.0902 - learning_rate: 0.0112
Epoch 63/100
10/10 — 0s 22ms/step - loss: 1.6896 - mae: 2.1391 - learning_rate: 0.0126
Epoch 64/100
10/10 — 0s 18ms/step - loss: 1.8600 - mae: 2.3135 - learning_rate: 0.0141
Epoch 65/100
10/10 — 0s 17ms/step - loss: 1.6616 - mae: 2.1058 - learning_rate: 0.0158
Epoch 66/100
10/10 — 0s 19ms/step - loss: 2.2572 - mae: 2.7217 - learning_rate: 0.0178
Epoch 67/100
10/10 — 0s 21ms/step - loss: 2.4561 - mae: 2.9251 - learning_rate: 0.0200
Epoch 68/100
10/10 — 0s 19ms/step - loss: 2.2592 - mae: 2.7198 - learning_rate: 0.0224
Epoch 69/100
10/10 — 0s 19ms/step - loss: 2.0799 - mae: 2.5379 - learning_rate: 0.0251
Epoch 70/100
10/10 — 0s 20ms/step - loss: 2.5298 - mae: 2.9947 - learning_rate: 0.0282
Epoch 71/100
10/10 — 0s 18ms/step - loss: 2.3009 - mae: 2.7616 - learning_rate: 0.0316
Epoch 72/100
10/10 — 0s 18ms/step - loss: 2.9574 - mae: 3.4238 - learning_rate: 0.0355
Epoch 73/100
10/10 — 0s 23ms/step - loss: 2.6127 - mae: 3.0749 - learning_rate: 0.0398
Epoch 74/100
10/10 — 0s 19ms/step - loss: 2.3569 - mae: 2.8195 - learning_rate: 0.0447
Epoch 75/100
10/10 — 0s 21ms/step - loss: 3.5991 - mae: 4.0742 - learning_rate: 0.0501
Epoch 76/100
10/10 — 0s 18ms/step - loss: 2.5871 - mae: 3.0524 - learning_rate: 0.0562
Epoch 77/100
10/10 — 0s 19ms/step - loss: 2.4278 - mae: 2.8885 - learning_rate: 0.0631
Epoch 78/100
10/10 — 0s 18ms/step - loss: 2.4250 - mae: 2.8901 - learning_rate: 0.0708

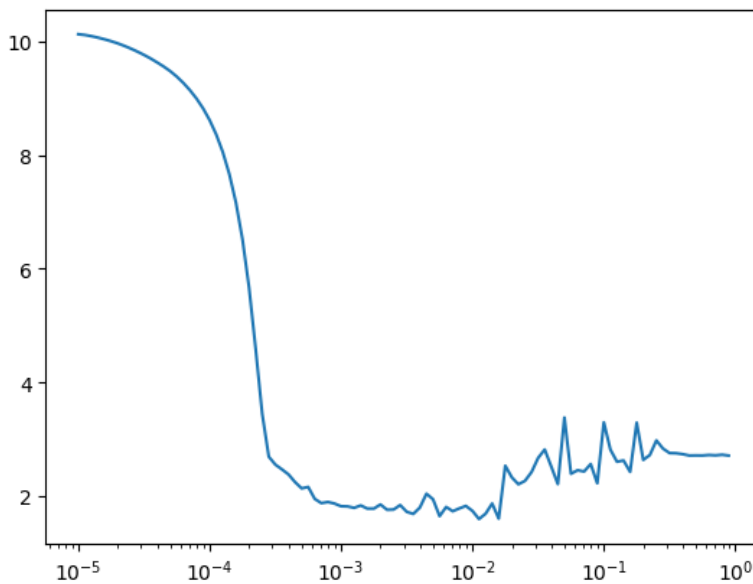

```

Epoch 79/100
10/10 — 0s 19ms/step - loss: 2.4784 - mae: 2.9446 - learning_rate: 0.0794
Epoch 80/100
10/10 — 0s 19ms/step - loss: 2.3932 - mae: 2.8553 - learning_rate: 0.0891
Epoch 81/100
10/10 — 0s 18ms/step - loss: 3.3934 - mae: 3.8695 - learning_rate: 0.1000
Epoch 82/100
10/10 — 0s 18ms/step - loss: 2.9790 - mae: 3.4454 - learning_rate: 0.1122
Epoch 83/100
10/10 — 0s 22ms/step - loss: 2.6124 - mae: 3.0776 - learning_rate: 0.1259
Epoch 84/100
10/10 — 0s 18ms/step - loss: 2.5110 - mae: 2.9773 - learning_rate: 0.1413
Epoch 85/100
10/10 — 0s 19ms/step - loss: 2.4846 - mae: 2.9530 - learning_rate: 0.1585
Epoch 86/100
10/10 — 0s 20ms/step - loss: 3.4071 - mae: 3.8808 - learning_rate: 0.1778
Epoch 87/100
10/10 — 0s 19ms/step - loss: 2.6264 - mae: 3.0948 - learning_rate: 0.1995
Epoch 88/100
10/10 — 0s 19ms/step - loss: 2.7250 - mae: 3.1922 - learning_rate: 0.2239
Epoch 89/100
10/10 — 0s 21ms/step - loss: 3.1129 - mae: 3.5824 - learning_rate: 0.2512
Epoch 90/100
10/10 — 0s 19ms/step - loss: 2.9334 - mae: 3.4009 - learning_rate: 0.2818
Epoch 91/100
10/10 — 0s 19ms/step - loss: 2.8676 - mae: 3.3379 - learning_rate: 0.3162
Epoch 92/100
10/10 — 0s 22ms/step - loss: 2.8186 - mae: 3.2858 - learning_rate: 0.3548
Epoch 93/100
10/10 — 0s 18ms/step - loss: 2.8369 - mae: 3.3054 - learning_rate: 0.3981
Epoch 94/100
10/10 — 0s 19ms/step - loss: 2.8013 - mae: 3.2671 - learning_rate: 0.4467
Epoch 95/100
10/10 — 0s 17ms/step - loss: 2.7902 - mae: 3.2589 - learning_rate: 0.5012
Epoch 96/100
10/10 — 0s 18ms/step - loss: 2.7834 - mae: 3.2498 - learning_rate: 0.5623
Epoch 97/100
10/10 — 0s 21ms/step - loss: 2.7956 - mae: 3.2624 - learning_rate: 0.6310
Epoch 98/100
10/10 — 0s 19ms/step - loss: 2.8098 - mae: 3.2776 - learning_rate: 0.7079
Epoch 99/100
10/10 — 0s 18ms/step - loss: 2.8556 - mae: 3.3239 - learning_rate: 0.7943
Epoch 100/100
10/10 — 0s 22ms/step - loss: 2.7791 - mae: 3.2460 - learning_rate: 0.8913

```

```
In [25]: plt.semilogx(lr_history.history["learning_rate"], lr_history.history["loss"])
```

```
Out[25]: [<matplotlib.lines.Line2D at 0x74d41867d950>]
```



Compiling the model

Exercise 3: 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 optional exercise, you should have a better idea of what a good learning rate would be.

Hints:

- 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 [57]: # 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.Huber(),
                  optimizer=tf.keras.optimizers.Adam(learning_rate=5e-4),
                  metrics=["mae"])

    ### END CODE HERE ###

    return model
```

```
In [58]: # Save an instance of the model
model = create_model()
```

```
In [59]: # Test your code!
unittests.test_create_model(create_model)

All tests passed!
```

If you passed the unittests, go ahead and train your model by running the cell below:

```
In [60]: # Train it
history = model.fit(train_dataset, epochs=50)
```

Epoch 1/50
10/10 ————— 2s 21ms/step - loss: 10.2620 - mae: 10.7608
Epoch 2/50
10/10 ————— 0s 23ms/step - loss: 9.3385 - mae: 9.8367
Epoch 3/50
10/10 ————— 0s 21ms/step - loss: 8.4829 - mae: 8.9807
Epoch 4/50
10/10 ————— 0s 22ms/step - loss: 7.5152 - mae: 8.0099
Epoch 5/50
10/10 ————— 0s 20ms/step - loss: 6.0641 - mae: 6.5528
Epoch 6/50
10/10 ————— 0s 19ms/step - loss: 4.8582 - mae: 5.3418
Epoch 7/50
10/10 ————— 0s 21ms/step - loss: 3.8309 - mae: 4.3063
Epoch 8/50
10/10 ————— 0s 19ms/step - loss: 3.1567 - mae: 3.6252
Epoch 9/50
10/10 ————— 0s 22ms/step - loss: 2.8046 - mae: 3.2718
Epoch 10/50
10/10 ————— 0s 19ms/step - loss: 2.8147 - mae: 3.2812
Epoch 11/50
10/10 ————— 0s 23ms/step - loss: 2.8306 - mae: 3.2979
Epoch 12/50
10/10 ————— 0s 20ms/step - loss: 2.7939 - mae: 3.2618
Epoch 13/50
10/10 ————— 0s 23ms/step - loss: 2.7790 - mae: 3.2469
Epoch 14/50
10/10 ————— 0s 20ms/step - loss: 2.6722 - mae: 3.1375
Epoch 15/50
10/10 ————— 0s 21ms/step - loss: 2.6537 - mae: 3.1194
Epoch 16/50
10/10 ————— 0s 20ms/step - loss: 2.6234 - mae: 3.0873
Epoch 17/50
10/10 ————— 0s 22ms/step - loss: 2.4873 - mae: 2.9503
Epoch 18/50
10/10 ————— 0s 20ms/step - loss: 2.4057 - mae: 2.8663
Epoch 19/50
10/10 ————— 0s 21ms/step - loss: 2.2287 - mae: 2.6867
Epoch 20/50
10/10 ————— 0s 20ms/step - loss: 2.0824 - mae: 2.5399
Epoch 21/50
10/10 ————— 0s 20ms/step - loss: 1.9849 - mae: 2.4370
Epoch 22/50
10/10 ————— 0s 21ms/step - loss: 1.8752 - mae: 2.3284
Epoch 23/50
10/10 ————— 0s 20ms/step - loss: 1.8634 - mae: 2.3144
Epoch 24/50
10/10 ————— 0s 23ms/step - loss: 1.7906 - mae: 2.2433
Epoch 25/50
10/10 ————— 0s 20ms/step - loss: 1.7569 - mae: 2.2033
Epoch 26/50
10/10 ————— 0s 19ms/step - loss: 1.8448 - mae: 2.2932
Epoch 27/50
10/10 ————— 0s 19ms/step - loss: 1.8071 - mae: 2.2579
Epoch 28/50
10/10 ————— 0s 21ms/step - loss: 1.7292 - mae: 2.1784
Epoch 29/50
10/10 ————— 0s 20ms/step - loss: 1.7447 - mae: 2.1926
Epoch 30/50
10/10 ————— 0s 21ms/step - loss: 1.7540 - mae: 2.2029
Epoch 31/50
10/10 ————— 0s 20ms/step - loss: 1.7203 - mae: 2.1712
Epoch 32/50
10/10 ————— 0s 21ms/step - loss: 1.7552 - mae: 2.2070
Epoch 33/50
10/10 ————— 0s 20ms/step - loss: 1.6904 - mae: 2.1386
Epoch 34/50
10/10 ————— 0s 22ms/step - loss: 1.6974 - mae: 2.1454
Epoch 35/50
10/10 ————— 0s 20ms/step - loss: 1.6515 - mae: 2.0973
Epoch 36/50
10/10 ————— 0s 22ms/step - loss: 1.6621 - mae: 2.1104
Epoch 37/50
10/10 ————— 0s 20ms/step - loss: 1.6905 - mae: 2.1405
Epoch 38/50
10/10 ————— 0s 20ms/step - loss: 1.6358 - mae: 2.0842
Epoch 39/50
10/10 ————— 0s 20ms/step - loss: 1.6463 - mae: 2.0927

```

Epoch 40/50
10/10 ————— 0s 24ms/step - loss: 1.6040 - mae: 2.0489
Epoch 41/50
10/10 ————— 0s 19ms/step - loss: 1.5549 - mae: 1.9967
Epoch 42/50
10/10 ————— 0s 19ms/step - loss: 1.5702 - mae: 2.0137
Epoch 43/50
10/10 ————— 0s 19ms/step - loss: 1.5463 - mae: 1.9878
Epoch 44/50
10/10 ————— 0s 22ms/step - loss: 1.5135 - mae: 1.9517
Epoch 45/50
10/10 ————— 0s 20ms/step - loss: 1.5299 - mae: 1.9708
Epoch 46/50
10/10 ————— 0s 22ms/step - loss: 1.5357 - mae: 1.9775
Epoch 47/50
10/10 ————— 0s 18ms/step - loss: 1.5047 - mae: 1.9434
Epoch 48/50
10/10 ————— 0s 19ms/step - loss: 1.5094 - mae: 1.9470
Epoch 49/50
10/10 ————— 0s 19ms/step - loss: 1.4794 - mae: 1.9134
Epoch 50/50
10/10 ————— 0s 23ms/step - loss: 1.4812 - mae: 1.9187

```

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

In [61]: *# Plot the training loss for each epoch*

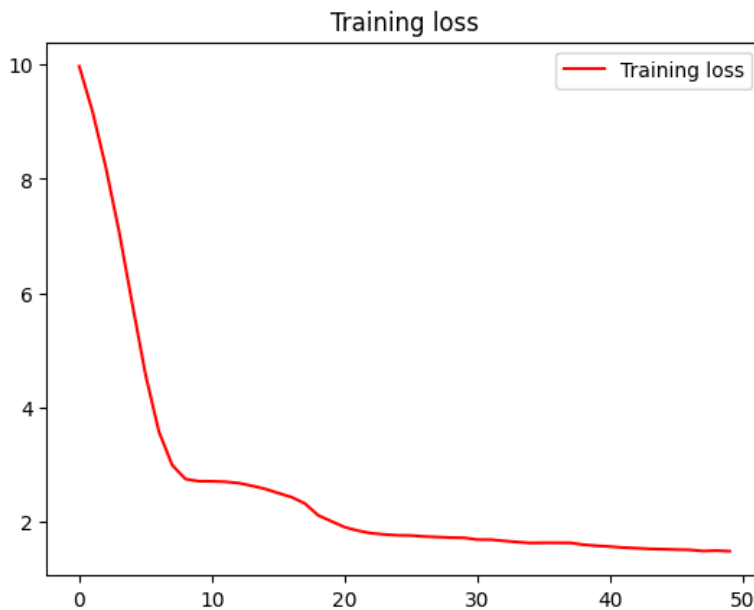
```

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 [62]: 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 only the model that will perform the forecast is ready but you still need to compute the actual forecast.

Faster model forecasts

In the previous weeks you used a for loop to compute the forecasts for every point in the sequence. This approach is valid but there is a more efficient way of doing the same thing by using batches of data. The code to implement this is provided in the `model_forecast` below. Notice that the code is very similar to the one in the `windowed_dataset` function with the differences that:

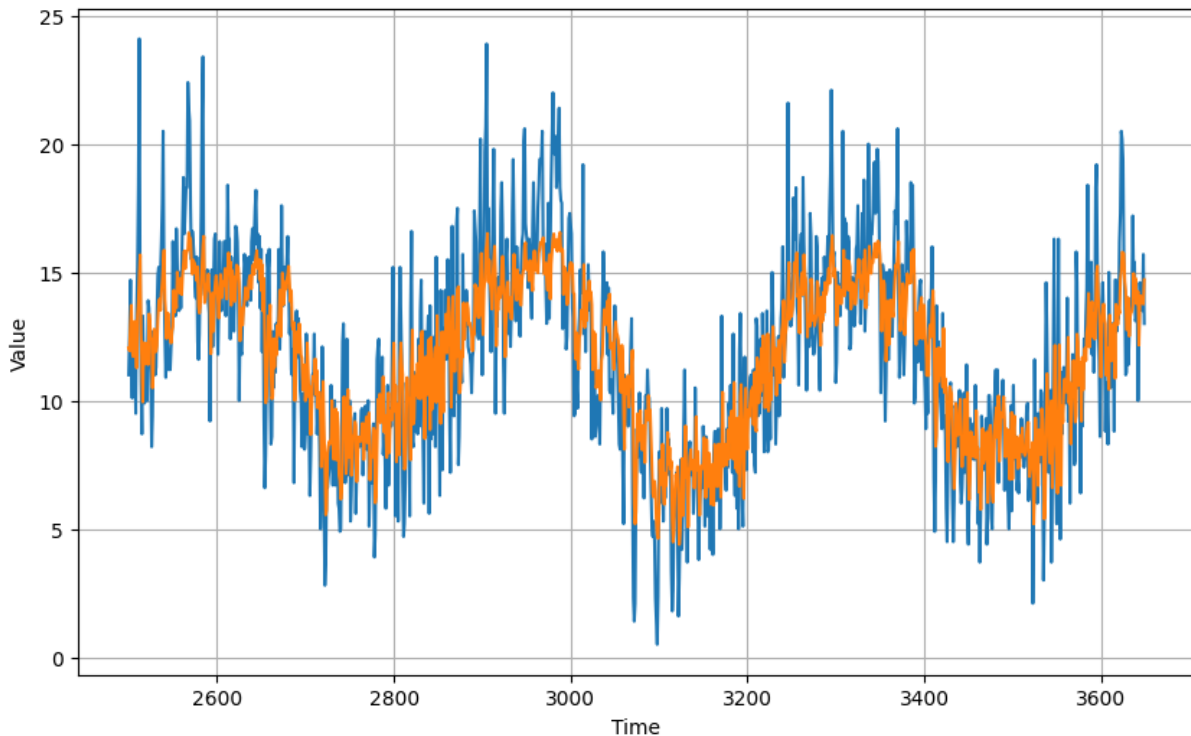
- The dataset is windowed using `window_size` rather than `window_size + 1`
- No shuffle should be used
- No need to split the data into features and labels
- A model is used to predict batches of the dataset

```
In [63]: def model_forecast(model, series, window_size):  
        """Generates a forecast using your trained model"""  
        ds = tf.data.Dataset.from_tensor_slices(series)  
        ds = ds.window(window_size, shift=1, drop_remainder=True)  
        ds = ds.flat_map(lambda w: w.batch(window_size))  
        ds = ds.batch(32).prefetch(1)  
        forecast = model.predict(ds)  
        return forecast
```

```
In [64]: # Compute the forecast for the validation dataset. Remember you need the last WINDOW_SIZE values to make the first prediction  
rnn_forecast = model_forecast(model, SERIES[SPLIT_TIME-WINDOW_SIZE:-1], WINDOW_SIZE).squeeze()
```

36/36 ————— 0s 7ms/step

```
In [65]: # Plot the forecast  
plt.figure(figsize=(10, 6))  
plot_series(time_valid, series_valid)  
plot_series(time_valid, rnn_forecast)
```



```
In [66]: mse, mae = compute_metrics(series_valid, rnn_forecast)
```

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

mse: 5.48, mae: 1.82 for forecast

To pass this assignment your forecast should achieve a MSE of 6 or less and a MAE of 2 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 these thresholds run the following cell to save the metrics in a binary file which will be used for grading. After

doing so, submit your assignment for grading.

```
In [67]: # Save metrics into a dictionary
metrics_dict = {
    "mse": float(mse),
    "mae": float(mae)
}

# Save your metrics in a binary file
with open('metrics.pkl', 'wb') as f:
    pickle.dump(metrics_dict, f)
```

Congratulations on finishing this week's assignment!

You have successfully implemented a neural network capable of forecasting time series leveraging a combination of Tensorflow's layers such as Convolutional and LSTMs! This resulted in a forecast that surpasses all the ones you did previously.

By finishing this assignment you have finished the specialization! Give yourself a pat on the back!!!