C4W3_Assignment

April 5, 2023

1 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.

Let's get started!

[1]:
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
```

1.1 Generating the data

The next cell includes a bunch of helper functions to generate and plot the time series:

[2]:
```
def plot_series(time, series, format="-", start=0, end=None):
    plt.plot(time[start:end], series[start:end], format)
    plt.xlabel("Time")
    plt.ylabel("Value")
    plt.grid(False)

def trend(time, slope=0):
    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):
    rnd = np.random.RandomState(seed)
```

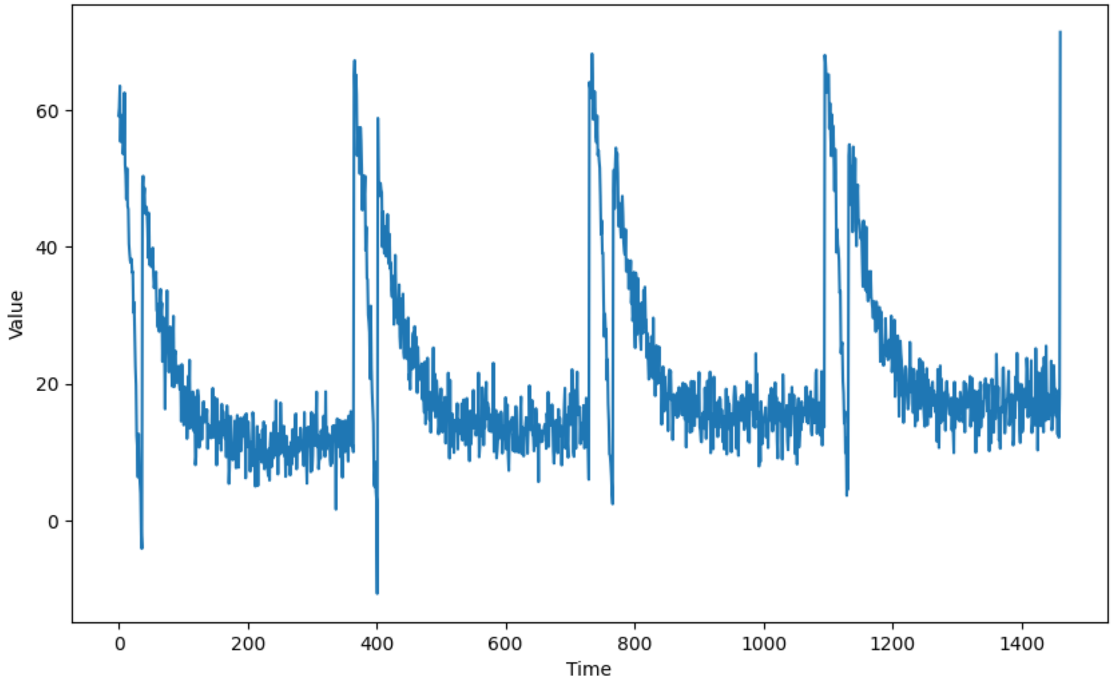
```
    return rnd.randn(len(time)) * noise_level
```

You will be generating the same time series data as in last week's assignment.

Notice that this time all the generation is done within a function and global variables are saved within a dataclass. This is done to avoid using global scope as it was done in during the first week of the course.

If you haven't used dataclasses before, they are just Python classes that provide a convenient syntax for storing data. You can read more about them in the docs.

[3]:
```
def generate_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


# Save all "global" variables within the G class (G stands for global)
@dataclass
class G:
    TIME, SERIES = generate_time_series()
    SPLIT_TIME = 1100
    WINDOW_SIZE = 20
    BATCH_SIZE = 32
    SHUFFLE_BUFFER_SIZE = 1000


# Plot the generated series
plt.figure(figsize=(10, 6))
plot_series(G.TIME, G.SERIES)
plt.show()
```



1.2 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:

[4]:
```
def train_val_split(time, series, time_step=G.SPLIT_TIME):

    time_train = time[:time_step]
    series_train = series[:time_step]
    time_valid = time[time_step:]
    series_valid = series[time_step:]

    return time_train, series_train, time_valid, series_valid


# Split the dataset
time_train, series_train, time_valid, series_valid = train_val_split(G.TIME, G.
 ↪SERIES)
```

[5]:
```
def windowed_dataset(series, window_size=G.WINDOW_SIZE, batch_size=G.
 ↪BATCH_SIZE, shuffle_buffer=G.SHUFFLE_BUFFER_SIZE):
    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)
```

```
    dataset = dataset.map(lambda window: (window[:-1], window[-1]))
    dataset = dataset.batch(batch_size).prefetch(1)
    return dataset

# Apply the transformation to the training set
dataset = windowed_dataset(series_train)
```

```
Metal device set to: Apple M1 Pro
```

```
2023-04-05 13:08:07.786452: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305]
Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel
may not have been built with NUMA support.
2023-04-05 13:08:07.787288: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271]
Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0
MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id:
<undefined>)
```

1.3 Defining the model architecture

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 you layer architecture. Unlike previous weeks or courses in which you define your layers and compile the model in the same function, here you will first need to complete 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: - Fill in the `Lambda` layers at the beginning and end of the network with the correct lamda functions. - 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.

[8]:
```
def create_uncompiled_model():

    ### START CODE HERE

    model = tf.keras.models.Sequential([
        tf.keras.layers.Lambda(lambda x: tf.expand_dims(x, axis=-1),
                               input_shape=[G.WINDOW_SIZE]),
        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)
    ])

    ### END CODE HERE

    return model
```

[9]:
```
# Test your uncompiled model
uncompiled_model = create_uncompiled_model()

try:
    uncompiled_model.predict(dataset)
except:
    print("Your current architecture is incompatible with the windowed dataset,␣
↪try adjusting it.")
else:
    print("Your current architecture is compatible with the windowed dataset! :
↪)")
```

```
2023-04-05 13:10:26.732399: W
tensorflow/core/platform/profile_utils/cpu_utils.cc:128] Failed to get CPU
frequency: 0 Hz
2023-04-05 13:10:26.796010: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113]
Plugin optimizer for device_type GPU is enabled.
```

```
34/34 [==============================] - 2s 47ms/step
Your current architecture is compatible with the windowed dataset! :)
```

1.4 Adjusting the learning rate - (Optional Exercise)

As you saw in the lecture 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 acommodates 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

[11]:
```
def adjust_learning_rate():

    model = create_uncompiled_model()

    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
```

```python
    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
```

[12]:
```python
# Run the training with dynamic LR
lr_history = adjust_learning_rate()
```

```
Epoch 1/100

2023-04-05 13:11:36.664883: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113]
Plugin optimizer for device_type GPU is enabled.

34/34 [==============================] - 12s 327ms/step - loss: 52.8408 - mae:
53.3393 - lr: 1.0000e-06
Epoch 2/100
34/34 [==============================] - 11s 323ms/step - loss: 9.5038 - mae:
9.9924 - lr: 1.1220e-06
Epoch 3/100
34/34 [==============================] - 11s 326ms/step - loss: 7.1741 - mae:
7.6574 - lr: 1.2589e-06
Epoch 4/100
34/34 [==============================] - 11s 323ms/step - loss: 6.2861 - mae:
6.7612 - lr: 1.4125e-06
Epoch 5/100
34/34 [==============================] - 11s 322ms/step - loss: 5.5708 - mae:
6.0493 - lr: 1.5849e-06
Epoch 6/100
34/34 [==============================] - 11s 324ms/step - loss: 4.8820 - mae:
5.3585 - lr: 1.7783e-06
Epoch 7/100
34/34 [==============================] - 11s 323ms/step - loss: 4.4834 - mae:
4.9570 - lr: 1.9953e-06
Epoch 8/100
34/34 [==============================] - 11s 322ms/step - loss: 4.2334 - mae:
4.7071 - lr: 2.2387e-06
Epoch 9/100
34/34 [==============================] - 11s 319ms/step - loss: 4.0407 - mae:
4.5065 - lr: 2.5119e-06
Epoch 10/100
34/34 [==============================] - 11s 322ms/step - loss: 3.8730 - mae:
4.3383 - lr: 2.8184e-06
```

```
Epoch 11/100
34/34 [==============================] - 11s 327ms/step - loss: 3.9228 - mae:
4.3923 - lr: 3.1623e-06
Epoch 12/100
34/34 [==============================] - 11s 330ms/step - loss: 4.2316 - mae:
4.7053 - lr: 3.5481e-06
Epoch 13/100
34/34 [==============================] - 11s 334ms/step - loss: 3.8423 - mae:
4.3123 - lr: 3.9811e-06
Epoch 14/100
34/34 [==============================] - 11s 334ms/step - loss: 3.7173 - mae:
4.1857 - lr: 4.4668e-06
Epoch 15/100
34/34 [==============================] - 12s 337ms/step - loss: 3.7852 - mae:
4.2569 - lr: 5.0119e-06
Epoch 16/100
34/34 [==============================] - 11s 336ms/step - loss: 3.6063 - mae:
4.0750 - lr: 5.6234e-06
Epoch 17/100
34/34 [==============================] - 12s 341ms/step - loss: 3.7536 - mae:
4.2209 - lr: 6.3096e-06
Epoch 18/100
34/34 [==============================] - 12s 338ms/step - loss: 3.3038 - mae:
3.7645 - lr: 7.0795e-06
Epoch 19/100
34/34 [==============================] - 11s 330ms/step - loss: 3.4473 - mae:
3.9169 - lr: 7.9433e-06
Epoch 20/100
34/34 [==============================] - 11s 328ms/step - loss: 3.5338 - mae:
4.0058 - lr: 8.9125e-06
Epoch 21/100
34/34 [==============================] - 11s 320ms/step - loss: 3.2539 - mae:
3.7227 - lr: 1.0000e-05
Epoch 22/100
34/34 [==============================] - 11s 330ms/step - loss: 3.7056 - mae:
4.1799 - lr: 1.1220e-05
Epoch 23/100
34/34 [==============================] - 11s 333ms/step - loss: 4.0493 - mae:
4.5222 - lr: 1.2589e-05
Epoch 24/100
34/34 [==============================] - 11s 333ms/step - loss: 3.4625 - mae:
3.9320 - lr: 1.4125e-05
Epoch 25/100
34/34 [==============================] - 11s 331ms/step - loss: 3.8655 - mae:
4.3392 - lr: 1.5849e-05
Epoch 26/100
34/34 [==============================] - 11s 330ms/step - loss: 3.4803 - mae:
3.9500 - lr: 1.7783e-05
```

```
Epoch 27/100
34/34 [==============================] - 11s 328ms/step - loss: 3.9427 - mae:
4.4152 - lr: 1.9953e-05
Epoch 28/100
34/34 [==============================] - 11s 327ms/step - loss: 3.5748 - mae:
4.0463 - lr: 2.2387e-05
Epoch 29/100
34/34 [==============================] - 12s 344ms/step - loss: 4.4924 - mae:
4.9733 - lr: 2.5119e-05
Epoch 30/100
34/34 [==============================] - 12s 341ms/step - loss: 7.1093 - mae:
7.5976 - lr: 2.8184e-05
Epoch 31/100
34/34 [==============================] - 12s 348ms/step - loss: 7.9796 - mae:
8.4703 - lr: 3.1623e-05
Epoch 32/100
34/34 [==============================] - 11s 334ms/step - loss: 3.8222 - mae:
4.2947 - lr: 3.5481e-05
Epoch 33/100
34/34 [==============================] - 12s 345ms/step - loss: 3.7212 - mae:
4.1911 - lr: 3.9811e-05
Epoch 34/100
34/34 [==============================] - 11s 328ms/step - loss: 8.7022 - mae:
9.1849 - lr: 4.4668e-05
Epoch 35/100
34/34 [==============================] - 11s 329ms/step - loss: 7.5319 - mae:
8.0200 - lr: 5.0119e-05
Epoch 36/100
34/34 [==============================] - 11s 331ms/step - loss: 6.5937 - mae:
7.0733 - lr: 5.6234e-05
Epoch 37/100
34/34 [==============================] - 11s 337ms/step - loss: 5.3198 - mae:
5.7944 - lr: 6.3096e-05
Epoch 38/100
34/34 [==============================] - 11s 326ms/step - loss: 8.6196 - mae:
9.1072 - lr: 7.0795e-05
Epoch 39/100
34/34 [==============================] - 11s 324ms/step - loss: 17.7174 - mae:
18.2155 - lr: 7.9433e-05
Epoch 40/100
34/34 [==============================] - 11s 326ms/step - loss: 14.3042 - mae:
14.7976 - lr: 8.9125e-05
Epoch 41/100
34/34 [==============================] - 11s 327ms/step - loss: 10.9648 - mae:
11.4587 - lr: 1.0000e-04
Epoch 42/100
34/34 [==============================] - 11s 325ms/step - loss: 16.2920 - mae:
16.7833 - lr: 1.1220e-04
```

```
Epoch 43/100
34/34 [==============================] - 11s 326ms/step - loss: 9.9508 - mae:
10.4377 - lr: 1.2589e-04
Epoch 44/100
34/34 [==============================] - 11s 326ms/step - loss: 20.3256 - mae:
20.8206 - lr: 1.4125e-04
Epoch 45/100
34/34 [==============================] - 11s 324ms/step - loss: 11.0859 - mae:
11.5769 - lr: 1.5849e-04
Epoch 46/100
34/34 [==============================] - 11s 329ms/step - loss: 9.5975 - mae:
10.0851 - lr: 1.7783e-04
Epoch 47/100
34/34 [==============================] - 11s 326ms/step - loss: 12.3006 - mae:
12.7939 - lr: 1.9953e-04
Epoch 48/100
34/34 [==============================] - 11s 324ms/step - loss: 8.6412 - mae:
9.1255 - lr: 2.2387e-04
Epoch 49/100
34/34 [==============================] - 11s 324ms/step - loss: 9.9237 - mae:
10.4130 - lr: 2.5119e-04
Epoch 50/100
34/34 [==============================] - 11s 328ms/step - loss: 11.4338 - mae:
11.9265 - lr: 2.8184e-04
Epoch 51/100
34/34 [==============================] - 11s 324ms/step - loss: 7.2142 - mae:
7.6970 - lr: 3.1623e-04
Epoch 52/100
34/34 [==============================] - 11s 333ms/step - loss: 13.7323 - mae:
14.2264 - lr: 3.5481e-04
Epoch 53/100
34/34 [==============================] - 11s 325ms/step - loss: 18.3479 - mae:
18.8438 - lr: 3.9811e-04
Epoch 54/100
34/34 [==============================] - 11s 329ms/step - loss: 6.0728 - mae:
6.5555 - lr: 4.4668e-04
Epoch 55/100
34/34 [==============================] - 11s 325ms/step - loss: 5.3170 - mae:
5.7969 - lr: 5.0119e-04
Epoch 56/100
34/34 [==============================] - 12s 338ms/step - loss: 7.0013 - mae:
7.4862 - lr: 5.6234e-04
Epoch 57/100
34/34 [==============================] - 12s 343ms/step - loss: 6.7593 - mae:
7.2437 - lr: 6.3096e-04
Epoch 58/100
34/34 [==============================] - 12s 364ms/step - loss: 5.8845 - mae:
6.3590 - lr: 7.0795e-04
```

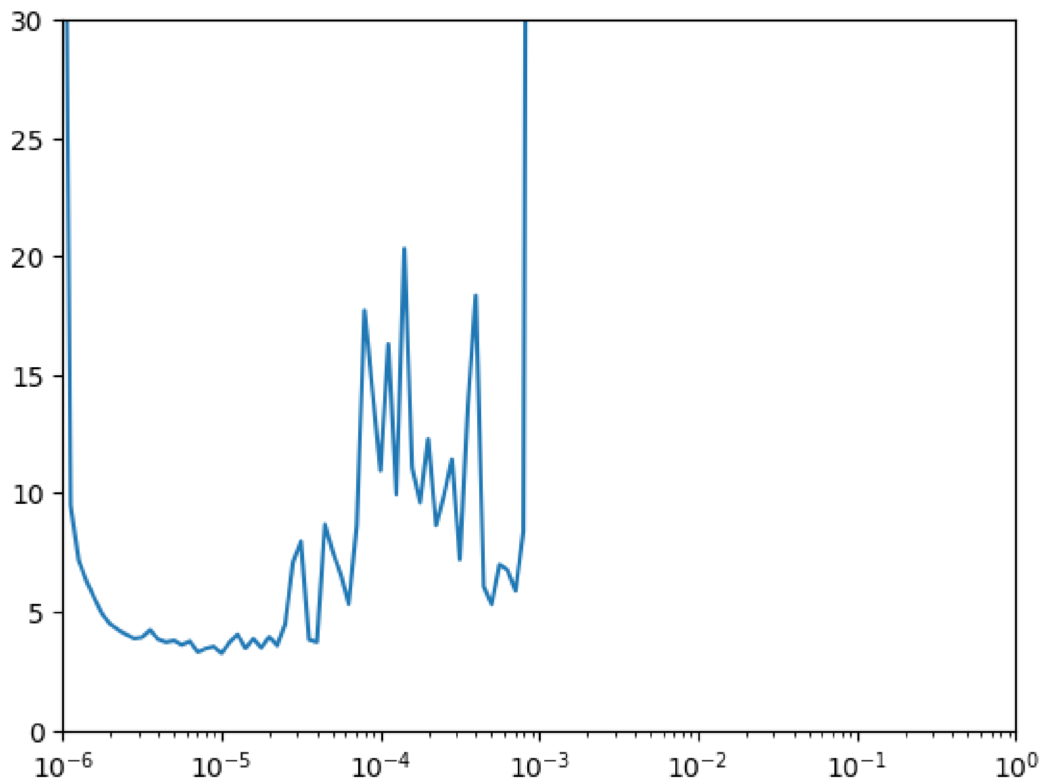
```
Epoch 59/100
34/34 [==============================] - 12s 344ms/step - loss: 8.3978 - mae:
8.8856 - lr: 7.9433e-04
Epoch 60/100
34/34 [==============================] - 11s 335ms/step - loss: 99.4819 - mae:
99.9819 - lr: 8.9125e-04
Epoch 61/100
34/34 [==============================] - 11s 335ms/step - loss: 117.0370 - mae:
117.5362 - lr: 0.0010
Epoch 62/100
34/34 [==============================] - 11s 332ms/step - loss: 358.1158 - mae:
358.6143 - lr: 0.0011
Epoch 63/100
34/34 [==============================] - 11s 334ms/step - loss: 498.2973 - mae:
498.7973 - lr: 0.0013
Epoch 64/100
34/34 [==============================] - 12s 366ms/step - loss: 449.2999 - mae:
449.7979 - lr: 0.0014
Epoch 65/100
34/34 [==============================] - 12s 341ms/step - loss: 333.4644 - mae:
333.9644 - lr: 0.0016
Epoch 66/100
34/34 [==============================] - 11s 336ms/step - loss: 681.7157 - mae:
682.2157 - lr: 0.0018
Epoch 67/100
34/34 [==============================] - 12s 350ms/step - loss: 1202.1833 - mae:
1202.6833 - lr: 0.0020
Epoch 68/100
34/34 [==============================] - 12s 342ms/step - loss: 488.4881 - mae:
488.9872 - lr: 0.0022
Epoch 69/100
34/34 [==============================] - 12s 363ms/step - loss: 751.0275 - mae:
751.5267 - lr: 0.0025
Epoch 70/100
34/34 [==============================] - 12s 353ms/step - loss: 1102.0396 - mae:
1102.5394 - lr: 0.0028
Epoch 71/100
34/34 [==============================] - 12s 344ms/step - loss: 1608.2765 - mae:
1608.7765 - lr: 0.0032
Epoch 72/100
34/34 [==============================] - 12s 362ms/step - loss: 742.8171 - mae:
743.3171 - lr: 0.0035
Epoch 73/100
34/34 [==============================] - 11s 331ms/step - loss: 844.6498 - mae:
845.1494 - lr: 0.0040
Epoch 74/100
34/34 [==============================] - 12s 352ms/step - loss: 2253.9783 - mae:
2254.4783 - lr: 0.0045
```

```
Epoch 75/100
34/34 [==============================] - 12s 357ms/step - loss: 1072.0190 - mae:
1072.5190 - lr: 0.0050
Epoch 76/100
34/34 [==============================] - 11s 337ms/step - loss: 2604.6980 - mae:
2605.1980 - lr: 0.0056
Epoch 77/100
34/34 [==============================] - 12s 344ms/step - loss: 4014.8606 - mae:
4015.3606 - lr: 0.0063
Epoch 78/100
34/34 [==============================] - 12s 344ms/step - loss: 2434.7856 - mae:
2435.2856 - lr: 0.0071
Epoch 79/100
34/34 [==============================] - 12s 339ms/step - loss: 4763.5381 - mae:
4764.0381 - lr: 0.0079
Epoch 80/100
34/34 [==============================] - 12s 339ms/step - loss: 2926.4180 - mae:
2926.9180 - lr: 0.0089
Epoch 81/100
34/34 [==============================] - 12s 341ms/step - loss: 4431.0117 - mae:
4431.5117 - lr: 0.0100
Epoch 82/100
34/34 [==============================] - 12s 351ms/step - loss: 6161.3115 - mae:
6161.8115 - lr: 0.0112
Epoch 83/100
34/34 [==============================] - 12s 345ms/step - loss: 3864.9109 - mae:
3865.4109 - lr: 0.0126
Epoch 84/100
34/34 [==============================] - 12s 367ms/step - loss: 7331.7271 - mae:
7332.2271 - lr: 0.0141
Epoch 85/100
34/34 [==============================] - 12s 360ms/step - loss: 10480.5166 -
mae: 10481.0166 - lr: 0.0158
Epoch 86/100
34/34 [==============================] - 12s 339ms/step - loss: 10117.8037 -
mae: 10118.3037 - lr: 0.0178
Epoch 87/100
34/34 [==============================] - 12s 344ms/step - loss: 8478.3203 - mae:
8478.8203 - lr: 0.0200
Epoch 88/100
34/34 [==============================] - 12s 359ms/step - loss: 11396.9414 -
mae: 11397.4414 - lr: 0.0224
Epoch 89/100
34/34 [==============================] - 12s 350ms/step - loss: 10399.0469 -
mae: 10399.5469 - lr: 0.0251
Epoch 90/100
34/34 [==============================] - 12s 363ms/step - loss: 18171.9297 -
mae: 18172.4297 - lr: 0.0282
```

```
Epoch 91/100
34/34 [==============================] - 13s 372ms/step - loss: 19767.6562 -
mae: 19768.1562 - lr: 0.0316
Epoch 92/100
34/34 [==============================] - 12s 353ms/step - loss: 17535.2734 -
mae: 17535.7734 - lr: 0.0355
Epoch 93/100
34/34 [==============================] - 12s 357ms/step - loss: 8417.9053 - mae:
8418.4053 - lr: 0.0398
Epoch 94/100
34/34 [==============================] - 13s 370ms/step - loss: 9509.0312 - mae:
9509.5312 - lr: 0.0447
Epoch 95/100
34/34 [==============================] - 12s 356ms/step - loss: 30228.9941 -
mae: 30229.4941 - lr: 0.0501
Epoch 96/100
34/34 [==============================] - 11s 335ms/step - loss: 36538.9492 -
mae: 36539.4492 - lr: 0.0562
Epoch 97/100
34/34 [==============================] - 11s 333ms/step - loss: 36878.0508 -
mae: 36878.5508 - lr: 0.0631
Epoch 98/100
34/34 [==============================] - 12s 349ms/step - loss: 38700.5039 -
mae: 38701.0039 - lr: 0.0708
Epoch 99/100
34/34 [==============================] - 12s 352ms/step - loss: 51588.4805 -
mae: 51588.9805 - lr: 0.0794
Epoch 100/100
34/34 [==============================] - 13s 370ms/step - loss: 56037.8203 -
mae: 56038.3203 - lr: 0.0891
```

[13]:
```
# Plot the loss for every LR
plt.semilogx(lr_history.history["lr"], lr_history.history["loss"])
plt.axis([1e-6, 1, 0, 30])
```

[13]:
```
(1e-06, 1.0, 0.0, 30.0)
```



1.5 Compiling the model

Now that you have trained the model while varying the learning rate, 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).

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.**

[14]:
```
def create_model():

    tf.random.set_seed(51)

    model = create_uncompiled_model()
```

```python
    ### START CODE HERE

    model.compile(loss=tf.keras.losses.Huber(),
                  optimizer=tf.keras.optimizers.SGD(learning_rate=1e-5,
 ↪momentum=0.9),
                  metrics=["mae"])

    ### END CODE HERE

    return model
```

[15]:
```python
# Save an instance of the model
model = create_model()

# Train it
history = model.fit(dataset, epochs=50)
```

```
Epoch 1/50

2023-04-05 13:31:10.081963: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113]
Plugin optimizer for device_type GPU is enabled.

34/34 [==============================] - 12s 333ms/step - loss: 14.0016 - mae:
14.4950
Epoch 2/50
34/34 [==============================] - 11s 330ms/step - loss: 7.3382 - mae:
7.8229
Epoch 3/50
34/34 [==============================] - 12s 339ms/step - loss: 5.0222 - mae:
5.5000
Epoch 4/50
34/34 [==============================] - 12s 347ms/step - loss: 5.4048 - mae:
5.8862
Epoch 5/50
34/34 [==============================] - 13s 375ms/step - loss: 3.8241 - mae:
4.2969
Epoch 6/50
34/34 [==============================] - 12s 367ms/step - loss: 3.6912 - mae:
4.1587
Epoch 7/50
34/34 [==============================] - 12s 351ms/step - loss: 3.6324 - mae:
4.1037
Epoch 8/50
34/34 [==============================] - 12s 353ms/step - loss: 4.8382 - mae:
5.3158
Epoch 9/50
34/34 [==============================] - 12s 348ms/step - loss: 3.8709 - mae:
```

```
4.3463
Epoch 10/50
34/34 [==============================] - 12s 348ms/step - loss: 3.4194 - mae:
3.8830
Epoch 11/50
34/34 [==============================] - 12s 352ms/step - loss: 3.6455 - mae:
4.1155
Epoch 12/50
34/34 [==============================] - 12s 362ms/step - loss: 3.3501 - mae:
3.8155
Epoch 13/50
34/34 [==============================] - 12s 357ms/step - loss: 4.2050 - mae:
4.6797
Epoch 14/50
34/34 [==============================] - 12s 340ms/step - loss: 3.6687 - mae:
4.1400
Epoch 15/50
34/34 [==============================] - 11s 332ms/step - loss: 3.3504 - mae:
3.8165
Epoch 16/50
34/34 [==============================] - 12s 351ms/step - loss: 3.1191 - mae:
3.5806
Epoch 17/50
34/34 [==============================] - 12s 361ms/step - loss: 3.7842 - mae:
4.2569
Epoch 18/50
34/34 [==============================] - 13s 371ms/step - loss: 3.4760 - mae:
3.9468
Epoch 19/50
34/34 [==============================] - 13s 371ms/step - loss: 3.0207 - mae:
3.4853
Epoch 20/50
34/34 [==============================] - 12s 350ms/step - loss: 4.5720 - mae:
5.0497
Epoch 21/50
34/34 [==============================] - 13s 383ms/step - loss: 3.1648 - mae:
3.6316
Epoch 22/50
34/34 [==============================] - 13s 374ms/step - loss: 3.4459 - mae:
3.9215
Epoch 23/50
34/34 [==============================] - 12s 337ms/step - loss: 3.4193 - mae:
3.8909
Epoch 24/50
34/34 [==============================] - 12s 351ms/step - loss: 3.7561 - mae:
4.2335
Epoch 25/50
34/34 [==============================] - 12s 355ms/step - loss: 2.8995 - mae:
```

```
3.3613
Epoch 26/50
34/34 [==============================] - 12s 345ms/step - loss: 3.0478 - mae:
3.5085
Epoch 27/50
34/34 [==============================] - 12s 342ms/step - loss: 3.0238 - mae:
3.4866
Epoch 28/50
34/34 [==============================] - 12s 343ms/step - loss: 3.1692 - mae:
3.6330
Epoch 29/50
34/34 [==============================] - 11s 333ms/step - loss: 3.5237 - mae:
4.0004
Epoch 30/50
34/34 [==============================] - 12s 337ms/step - loss: 3.1058 - mae:
3.5668
Epoch 31/50
34/34 [==============================] - 12s 345ms/step - loss: 3.4515 - mae:
3.9260
Epoch 32/50
34/34 [==============================] - 12s 350ms/step - loss: 3.0037 - mae:
3.4659
Epoch 33/50
34/34 [==============================] - 12s 359ms/step - loss: 3.2645 - mae:
3.7303
Epoch 34/50
34/34 [==============================] - 12s 346ms/step - loss: 2.9715 - mae:
3.4394
Epoch 35/50
34/34 [==============================] - 12s 351ms/step - loss: 3.0791 - mae:
3.5482
Epoch 36/50
34/34 [==============================] - 13s 372ms/step - loss: 3.4602 - mae:
3.9317
Epoch 37/50
34/34 [==============================] - 11s 334ms/step - loss: 3.6713 - mae:
4.1454
Epoch 38/50
34/34 [==============================] - 12s 340ms/step - loss: 2.7917 - mae:
3.2524
Epoch 39/50
34/34 [==============================] - 11s 332ms/step - loss: 3.1651 - mae:
3.6343
Epoch 40/50
34/34 [==============================] - 11s 332ms/step - loss: 2.9492 - mae:
3.4122
Epoch 41/50
34/34 [==============================] - 11s 329ms/step - loss: 3.2777 - mae:
```

```
3.7503
Epoch 42/50
34/34 [==============================] - 11s 331ms/step - loss: 2.8607 - mae:
3.3242
Epoch 43/50
34/34 [==============================] - 11s 332ms/step - loss: 2.9085 - mae:
3.3733
Epoch 44/50
34/34 [==============================] - 11s 329ms/step - loss: 2.9403 - mae:
3.4039
Epoch 45/50
34/34 [==============================] - 11s 334ms/step - loss: 2.9266 - mae:
3.3929
Epoch 46/50
34/34 [==============================] - 11s 331ms/step - loss: 2.8197 - mae:
3.2794
Epoch 47/50
34/34 [==============================] - 11s 332ms/step - loss: 2.9003 - mae:
3.3609
Epoch 48/50
34/34 [==============================] - 11s 331ms/step - loss: 3.4216 - mae:
3.8901
Epoch 49/50
34/34 [==============================] - 11s 333ms/step - loss: 2.9249 - mae:
3.3901
Epoch 50/50
34/34 [==============================] - 11s 335ms/step - loss: 3.0177 - mae:
3.4828
```

1.6 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:

[16]:
```python
def compute_metrics(true_series, forecast):

    mse = tf.keras.metrics.mean_squared_error(true_series, forecast).numpy()
    mae = tf.keras.metrics.mean_absolute_error(true_series, forecast).numpy()

    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.

1.7 Faster model forecasts

In the previous week 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

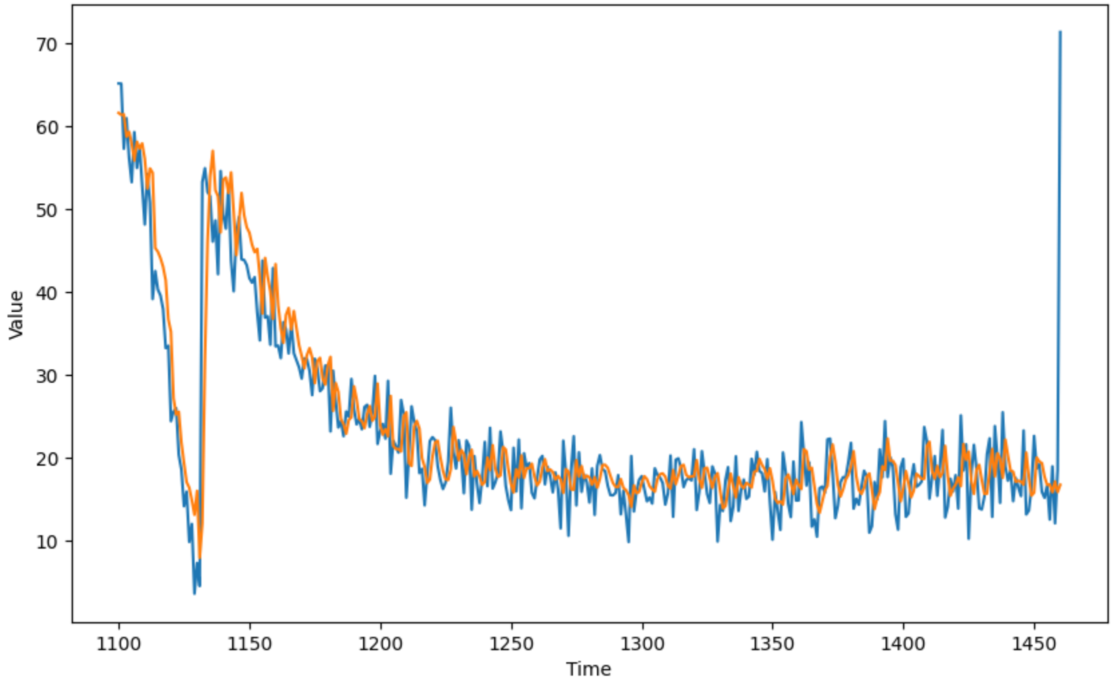
[17]:
```
def model_forecast(model, series, window_size):
    ds = tf.data.Dataset.from_tensor_slices(series)
    ds = ds.window(window_size, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda w: w.batch(window_size))
    ds = ds.batch(32).prefetch(1)
    forecast = model.predict(ds)
    return forecast
```

[18]:
```
# Compute the forecast for all the series
rnn_forecast = model_forecast(model, G.SERIES, G.WINDOW_SIZE).squeeze()

# Slice the forecast to get only the predictions for the validation set
rnn_forecast = rnn_forecast[G.SPLIT_TIME - G.WINDOW_SIZE:-1]

# Plot it
plt.figure(figsize=(10, 6))

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

```
      3/Unknown - 0s 45ms/step
```

```
2023-04-05 13:41:01.543958: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:113]
Plugin optimizer for device_type GPU is enabled.
```

```
46/46 [==============================] - 2s 47ms/step
```



Expected Output:

A series similar to this one:

[19]:
```
mse, mae = compute_metrics(series_valid, rnn_forecast)

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

```
mse: 31.85, mae: 3.64 for forecast
```

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 model in a `tar` file which will be used for grading and after doing so, submit your assigment for grading.
- This environment includes a dummy `SavedModel` directory which contains a dummy model trained for one epoch. **To replace this file with your actual model you need to run the next cell before submitting for grading.**
- Unlike last week, this time the model is saved using the `SavedModel` format. This is done because the HDF5 format does not fully support `Lambda` layers.

[20]:
```
# Save your model in the SavedModel format
model.save('saved_model/my_model')
```

```
# Compress the directory using tar
! tar -czvf saved_model.tar.gz saved_model/
```

```
INFO:tensorflow:Assets written to: saved_model/my_model/assets
a saved_model
a saved_model/my_model
a saved_model/my_model/keras_metadata.pb
a saved_model/my_model/variables
a saved_model/my_model/saved_model.pb
a saved_model/my_model/assets
a saved_model/my_model/variables/variables.data-00000-of-00001
a saved_model/my_model/variables/variables.index
```

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!