C4W4_Assignment

November 9, 2022

1 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.

Let's get started!

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

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

[2]:
```
TEMPERATURES_CSV = './data/daily-min-temperatures.csv'

with open(TEMPERATURES_CSV, '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.

[3]:
```
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(True)
```

1.1 Parsing the raw data

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

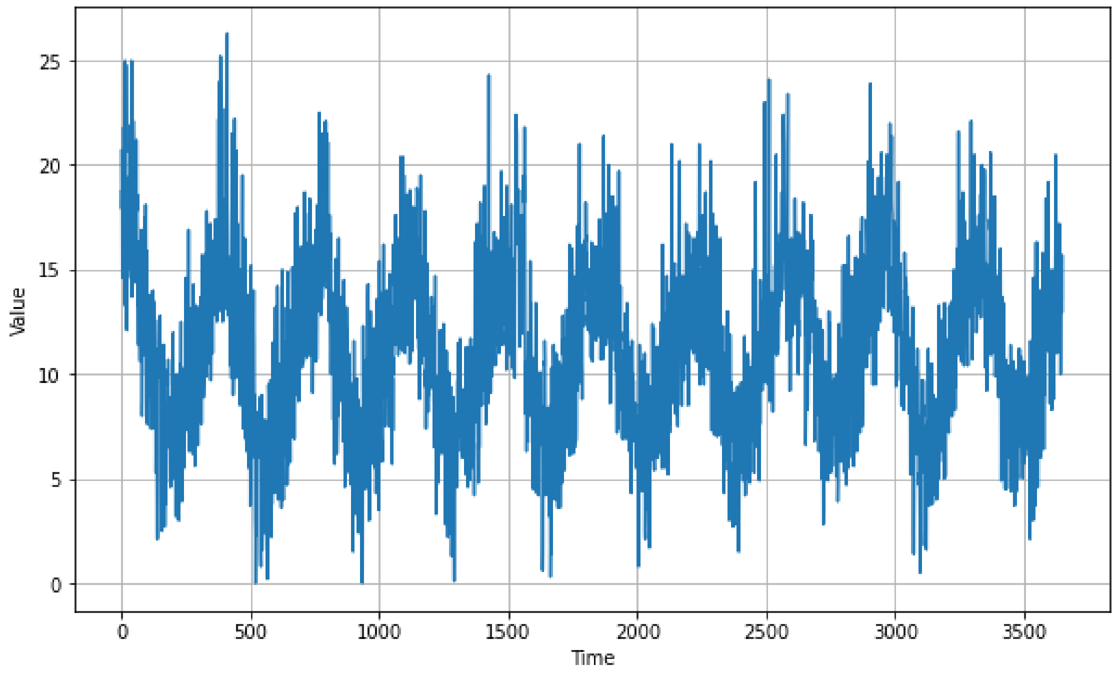
A couple of things to note:

- You should omit the first line as the file contains headers.
- There is no need to save the data points as numpy arrays, regular lists is fine.
- To read from csv files use `csv.reader` by passing the appropriate arguments.
- `csv.reader` returns an iterable that returns each row in every iteration. So the temperature can be accessed via row[1] and the date can be discarded.
- 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. You can use Python's built-in `float` function to ensure this.

[4]:
```
def parse_data_from_file(filename):

    times = []
    temperatures = []

    with open(filename) as csvfile:

        ### START CODE HERE

        reader = csv.reader(csvfile, delimiter=',')
        next(reader)
        for index, row in enumerate(reader):
            times.append(index)
            temperatures.append(float(row[1]))
        ### END CODE HERE

    return times, temperatures
```

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:

```
[5]: # Test your function and save all "global" variables within the G class (G␣
     ↪stands for global)
     @dataclass
     class G:
         TEMPERATURES_CSV = './data/daily-min-temperatures.csv'
         times, temperatures = parse_data_from_file(TEMPERATURES_CSV)
         TIME = np.array(times)
         SERIES = np.array(temperatures)
         SPLIT_TIME = 2500
         WINDOW_SIZE = 64
         BATCH_SIZE = 256
         SHUFFLE_BUFFER_SIZE = 1000


     plt.figure(figsize=(10, 6))
     plot_series(G.TIME, G.SERIES)
     plt.show()
```



Expected Output:

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:

[6]:
```
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)
```

[7]:
```
def windowed_dataset(series, window_size=G.WINDOW_SIZE, batch_size=G.
 ↪BATCH_SIZE, shuffle_buffer=G.SHUFFLE_BUFFER_SIZE):
    ds = tf.data.Dataset.from_tensor_slices(series)
    ds = ds.window(window_size + 1, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda w: w.batch(window_size + 1))
    ds = ds.shuffle(shuffle_buffer)
    ds = ds.map(lambda w: (w[:-1], w[-1]))
    ds = ds.batch(batch_size).prefetch(1)
    return ds


# Apply the transformation to the training set
train_set = windowed_dataset(series_train, window_size=G.WINDOW_SIZE,␣
 ↪batch_size=G.BATCH_SIZE, shuffle_buffer=G.SHUFFLE_BUFFER_SIZE)
```

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 your layer 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:

- `Lambda` layers are not required.
- Use a combination of `Conv1D` and `LSTM` layers followed by `Dense` layers

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

    ### START CODE HERE

    model = tf.keras.models.Sequential([
        tf.keras.layers.Conv1D(
            filters=64,
            kernel_size=3,
            strides=1,
            activation='relu',
            padding='causal',
            input_shape=[G.WINDOW_SIZE, 1]
        ),
        tf.keras.layers.LSTM(64, return_sequences=True),
        tf.keras.layers.LSTM(64),
        tf.keras.layers.Dense(30, activation='relu'),
        tf.keras.layers.Dense(10, activation='relu'),
        tf.keras.layers.Dense(1),

    ])

    ### END CODE HERE

    return model
```

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

try:
    uncompiled_model.predict(train_set)
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! :
 ↪)")
```

```
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

[10]:
```
def adjust_learning_rate(dataset):

    model = create_uncompiled_model()

    lr_schedule = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-4 *␣
 ↪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
```

[11]:
```
# Run the training with dynamic LR
lr_history = adjust_learning_rate(train_set)
```

```
Epoch 1/100
10/10 [==============================] - 7s 358ms/step - loss: 10.3521 - mae:
10.8514 - lr: 1.0000e-04
Epoch 2/100
10/10 [==============================] - 4s 334ms/step - loss: 10.3200 - mae:
10.8193 - lr: 1.1220e-04
Epoch 3/100
10/10 [==============================] - 3s 323ms/step - loss: 10.2741 - mae:
10.7732 - lr: 1.2589e-04
Epoch 4/100
10/10 [==============================] - 3s 312ms/step - loss: 10.2215 - mae:
10.7205 - lr: 1.4125e-04
Epoch 5/100
10/10 [==============================] - 3s 323ms/step - loss: 10.1697 - mae:
10.6686 - lr: 1.5849e-04
Epoch 6/100
10/10 [==============================] - 3s 322ms/step - loss: 10.1341 - mae:
10.6329 - lr: 1.7783e-04
Epoch 7/100
10/10 [==============================] - 3s 322ms/step - loss: 10.1001 - mae:
10.5989 - lr: 1.9953e-04
```

```
Epoch 8/100
10/10 [==============================] - 3s 321ms/step - loss: 10.0580 - mae:
10.5568 - lr: 2.2387e-04
Epoch 9/100
10/10 [==============================] - 3s 322ms/step - loss: 10.0066 - mae:
10.5053 - lr: 2.5119e-04
Epoch 10/100
10/10 [==============================] - 3s 322ms/step - loss: 9.9408 - mae:
10.4394 - lr: 2.8184e-04
Epoch 11/100
10/10 [==============================] - 3s 321ms/step - loss: 9.8507 - mae:
10.3493 - lr: 3.1623e-04
Epoch 12/100
10/10 [==============================] - 3s 313ms/step - loss: 9.7310 - mae:
10.2295 - lr: 3.5481e-04
Epoch 13/100
10/10 [==============================] - 3s 311ms/step - loss: 9.5579 - mae:
10.0564 - lr: 3.9811e-04
Epoch 14/100
10/10 [==============================] - 3s 323ms/step - loss: 9.2989 - mae:
9.7971 - lr: 4.4668e-04
Epoch 15/100
10/10 [==============================] - 3s 321ms/step - loss: 8.8905 - mae:
9.3887 - lr: 5.0119e-04
Epoch 16/100
10/10 [==============================] - 3s 323ms/step - loss: 8.2008 - mae:
8.6983 - lr: 5.6234e-04
Epoch 17/100
10/10 [==============================] - 3s 320ms/step - loss: 6.9207 - mae:
7.4156 - lr: 6.3096e-04
Epoch 18/100
10/10 [==============================] - 3s 320ms/step - loss: 4.5434 - mae:
5.0237 - lr: 7.0795e-04
Epoch 19/100
10/10 [==============================] - 3s 323ms/step - loss: 2.7128 - mae:
3.1819 - lr: 7.9433e-04
Epoch 20/100
10/10 [==============================] - 3s 324ms/step - loss: 2.6091 - mae:
3.0755 - lr: 8.9125e-04
Epoch 21/100
10/10 [==============================] - 3s 322ms/step - loss: 2.4584 - mae:
2.9199 - lr: 0.0010
Epoch 22/100
10/10 [==============================] - 3s 313ms/step - loss: 2.1510 - mae:
2.6081 - lr: 0.0011
Epoch 23/100
10/10 [==============================] - 3s 322ms/step - loss: 1.8742 - mae:
2.3277 - lr: 0.0013
```

```
Epoch 24/100
10/10 [==============================] - 3s 322ms/step - loss: 2.0155 - mae:
2.4704 - lr: 0.0014
Epoch 25/100
10/10 [==============================] - 3s 315ms/step - loss: 2.0222 - mae:
2.4804 - lr: 0.0016
Epoch 26/100
10/10 [==============================] - 3s 321ms/step - loss: 1.8683 - mae:
2.3210 - lr: 0.0018
Epoch 27/100
10/10 [==============================] - 3s 313ms/step - loss: 1.7858 - mae:
2.2363 - lr: 0.0020
Epoch 28/100
10/10 [==============================] - 3s 321ms/step - loss: 1.7433 - mae:
2.1947 - lr: 0.0022
Epoch 29/100
10/10 [==============================] - 3s 313ms/step - loss: 1.8658 - mae:
2.3187 - lr: 0.0025
Epoch 30/100
10/10 [==============================] - 3s 320ms/step - loss: 1.8670 - mae:
2.3209 - lr: 0.0028
Epoch 31/100
10/10 [==============================] - 3s 312ms/step - loss: 1.7403 - mae:
2.1924 - lr: 0.0032
Epoch 32/100
10/10 [==============================] - 3s 322ms/step - loss: 1.8416 - mae:
2.2962 - lr: 0.0035
Epoch 33/100
10/10 [==============================] - 3s 314ms/step - loss: 1.7479 - mae:
2.1990 - lr: 0.0040
Epoch 34/100
10/10 [==============================] - 3s 322ms/step - loss: 1.9010 - mae:
2.3569 - lr: 0.0045
Epoch 35/100
10/10 [==============================] - 3s 320ms/step - loss: 1.7618 - mae:
2.2127 - lr: 0.0050
Epoch 36/100
10/10 [==============================] - 3s 313ms/step - loss: 1.6072 - mae:
2.0516 - lr: 0.0056
Epoch 37/100
10/10 [==============================] - 3s 324ms/step - loss: 1.6663 - mae:
2.1140 - lr: 0.0063
Epoch 38/100
10/10 [==============================] - 3s 313ms/step - loss: 1.7042 - mae:
2.1530 - lr: 0.0071
Epoch 39/100
10/10 [==============================] - 3s 313ms/step - loss: 1.6136 - mae:
2.0576 - lr: 0.0079
```

```
Epoch 40/100
10/10 [==============================] - 3s 312ms/step - loss: 1.6104 - mae:
2.0571 - lr: 0.0089
Epoch 41/100
10/10 [==============================] - 3s 320ms/step - loss: 1.7093 - mae:
2.1548 - lr: 0.0100
Epoch 42/100
10/10 [==============================] - 3s 321ms/step - loss: 1.9608 - mae:
2.4180 - lr: 0.0112
Epoch 43/100
10/10 [==============================] - 3s 312ms/step - loss: 1.8287 - mae:
2.2783 - lr: 0.0126
Epoch 44/100
10/10 [==============================] - 3s 322ms/step - loss: 1.8216 - mae:
2.2738 - lr: 0.0141
Epoch 45/100
10/10 [==============================] - 3s 320ms/step - loss: 1.8182 - mae:
2.2690 - lr: 0.0158
Epoch 46/100
10/10 [==============================] - 3s 332ms/step - loss: 1.8530 - mae:
2.3109 - lr: 0.0178
Epoch 47/100
10/10 [==============================] - 3s 320ms/step - loss: 1.9566 - mae:
2.4129 - lr: 0.0200
Epoch 48/100
10/10 [==============================] - 3s 312ms/step - loss: 1.9500 - mae:
2.4068 - lr: 0.0224
Epoch 49/100
10/10 [==============================] - 3s 323ms/step - loss: 2.2353 - mae:
2.6960 - lr: 0.0251
Epoch 50/100
10/10 [==============================] - 3s 321ms/step - loss: 2.3649 - mae:
2.8275 - lr: 0.0282
Epoch 51/100
10/10 [==============================] - 3s 320ms/step - loss: 2.2376 - mae:
2.6983 - lr: 0.0316
Epoch 52/100
10/10 [==============================] - 3s 322ms/step - loss: 2.3025 - mae:
2.7569 - lr: 0.0355
Epoch 53/100
10/10 [==============================] - 3s 322ms/step - loss: 2.4794 - mae:
2.9439 - lr: 0.0398
Epoch 54/100
10/10 [==============================] - 3s 323ms/step - loss: 2.1442 - mae:
2.6012 - lr: 0.0447
Epoch 55/100
10/10 [==============================] - 3s 332ms/step - loss: 3.1302 - mae:
3.6009 - lr: 0.0501
```

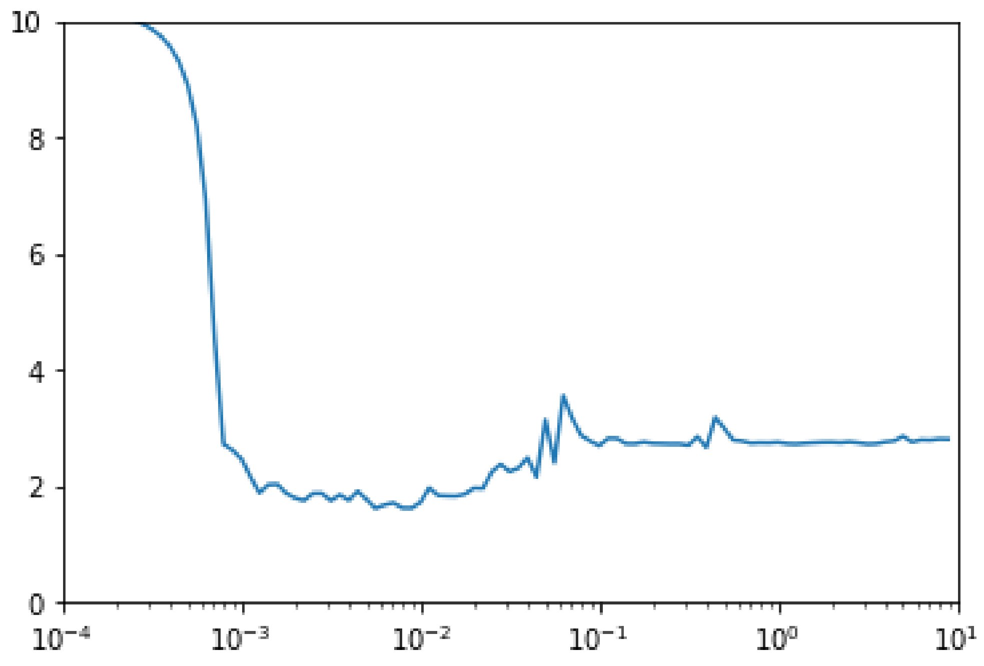
```
Epoch 56/100
10/10 [==============================] - 3s 320ms/step - loss: 2.3957 - mae:
2.8596 - lr: 0.0562
Epoch 57/100
10/10 [==============================] - 3s 313ms/step - loss: 3.5431 - mae:
4.0157 - lr: 0.0631
Epoch 58/100
10/10 [==============================] - 3s 314ms/step - loss: 3.1559 - mae:
3.6295 - lr: 0.0708
Epoch 59/100
10/10 [==============================] - 3s 323ms/step - loss: 2.8610 - mae:
3.3270 - lr: 0.0794
Epoch 60/100
10/10 [==============================] - 3s 322ms/step - loss: 2.7627 - mae:
3.2275 - lr: 0.0891
Epoch 61/100
10/10 [==============================] - 3s 320ms/step - loss: 2.6855 - mae:
3.1523 - lr: 0.1000
Epoch 62/100
10/10 [==============================] - 3s 322ms/step - loss: 2.8099 - mae:
3.2777 - lr: 0.1122
Epoch 63/100
10/10 [==============================] - 3s 322ms/step - loss: 2.8081 - mae:
3.2768 - lr: 0.1259
Epoch 64/100
10/10 [==============================] - 3s 323ms/step - loss: 2.7189 - mae:
3.1862 - lr: 0.1413
Epoch 65/100
10/10 [==============================] - 3s 311ms/step - loss: 2.7173 - mae:
3.1842 - lr: 0.1585
Epoch 66/100
10/10 [==============================] - 3s 320ms/step - loss: 2.7481 - mae:
3.2145 - lr: 0.1778
Epoch 67/100
10/10 [==============================] - 3s 322ms/step - loss: 2.7231 - mae:
3.1916 - lr: 0.1995
Epoch 68/100
10/10 [==============================] - 3s 320ms/step - loss: 2.7195 - mae:
3.1853 - lr: 0.2239
Epoch 69/100
10/10 [==============================] - 3s 323ms/step - loss: 2.7144 - mae:
3.1818 - lr: 0.2512
Epoch 70/100
10/10 [==============================] - 3s 321ms/step - loss: 2.7154 - mae:
3.1800 - lr: 0.2818
Epoch 71/100
10/10 [==============================] - 3s 322ms/step - loss: 2.6925 - mae:
3.1576 - lr: 0.3162
```

```
Epoch 72/100
10/10 [==============================] - 3s 323ms/step - loss: 2.8373 - mae:
3.3065 - lr: 0.3548
Epoch 73/100
10/10 [==============================] - 3s 313ms/step - loss: 2.6564 - mae:
3.1223 - lr: 0.3981
Epoch 74/100
10/10 [==============================] - 3s 322ms/step - loss: 3.1735 - mae:
3.6467 - lr: 0.4467
Epoch 75/100
10/10 [==============================] - 3s 321ms/step - loss: 2.9832 - mae:
3.4536 - lr: 0.5012
Epoch 76/100
10/10 [==============================] - 3s 320ms/step - loss: 2.7817 - mae:
3.2481 - lr: 0.5623
Epoch 77/100
10/10 [==============================] - 3s 322ms/step - loss: 2.7644 - mae:
3.2324 - lr: 0.6310
Epoch 78/100
10/10 [==============================] - 3s 322ms/step - loss: 2.7278 - mae:
3.1924 - lr: 0.7079
Epoch 79/100
10/10 [==============================] - 3s 322ms/step - loss: 2.7356 - mae:
3.2030 - lr: 0.7943
Epoch 80/100
10/10 [==============================] - 3s 321ms/step - loss: 2.7325 - mae:
3.1997 - lr: 0.8913
Epoch 81/100
10/10 [==============================] - 3s 324ms/step - loss: 2.7441 - mae:
3.2114 - lr: 1.0000
Epoch 82/100
10/10 [==============================] - 3s 323ms/step - loss: 2.7193 - mae:
3.1862 - lr: 1.1220
Epoch 83/100
10/10 [==============================] - 3s 312ms/step - loss: 2.7132 - mae:
3.1797 - lr: 1.2589
Epoch 84/100
10/10 [==============================] - 3s 314ms/step - loss: 2.7259 - mae:
3.1919 - lr: 1.4125
Epoch 85/100
10/10 [==============================] - 3s 313ms/step - loss: 2.7354 - mae:
3.2019 - lr: 1.5849
Epoch 86/100
10/10 [==============================] - 3s 323ms/step - loss: 2.7437 - mae:
3.2105 - lr: 1.7783
Epoch 87/100
10/10 [==============================] - 3s 323ms/step - loss: 2.7453 - mae:
3.2130 - lr: 1.9953
```

```
Epoch 88/100
10/10 [==============================] - 3s 322ms/step - loss: 2.7368 - mae:
3.2024 - lr: 2.2387
Epoch 89/100
10/10 [==============================] - 3s 321ms/step - loss: 2.7514 - mae:
3.2172 - lr: 2.5119
Epoch 90/100
10/10 [==============================] - 3s 313ms/step - loss: 2.7332 - mae:
3.2001 - lr: 2.8184
Epoch 91/100
10/10 [==============================] - 3s 323ms/step - loss: 2.7128 - mae:
3.1780 - lr: 3.1623
Epoch 92/100
10/10 [==============================] - 3s 311ms/step - loss: 2.7183 - mae:
3.1852 - lr: 3.5481
Epoch 93/100
10/10 [==============================] - 3s 313ms/step - loss: 2.7493 - mae:
3.2168 - lr: 3.9811
Epoch 94/100
10/10 [==============================] - 3s 322ms/step - loss: 2.7669 - mae:
3.2332 - lr: 4.4668
Epoch 95/100
10/10 [==============================] - 3s 322ms/step - loss: 2.8467 - mae:
3.3185 - lr: 5.0119
Epoch 96/100
10/10 [==============================] - 3s 321ms/step - loss: 2.7474 - mae:
3.2143 - lr: 5.6234
Epoch 97/100
10/10 [==============================] - 3s 314ms/step - loss: 2.7859 - mae:
3.2542 - lr: 6.3096
Epoch 98/100
10/10 [==============================] - 3s 321ms/step - loss: 2.7800 - mae:
3.2507 - lr: 7.0795
Epoch 99/100
10/10 [==============================] - 3s 324ms/step - loss: 2.7997 - mae:
3.2655 - lr: 7.9433
Epoch 100/100
10/10 [==============================] - 3s 314ms/step - loss: 2.7966 - mae:
3.2670 - lr: 8.9125
```

[12]:
```
plt.semilogx(lr_history.history["lr"], lr_history.history["loss"])
plt.axis([1e-4, 10, 0, 10])
```

[12]:
```
(0.0001, 10.0, 0.0, 10.0)
```



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).

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.

[13]:
```python
def create_model():


    model = create_uncompiled_model()

    ### START CODE HERE

    model.compile(loss=tf.keras.losses.Huber(),
                  optimizer=tf.keras.optimizers.SGD(learning_rate=4e-3,␣
↪momentum=0.9),
                  metrics=["mae"])


    ### END CODE HERE

    return model
```

[14]:
```
# Save an instance of the model
model = create_model()

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

```
Epoch 1/50
10/10 [==============================] - 6s 325ms/step - loss: 9.4275 - mae:
9.9254
Epoch 2/50
10/10 [==============================] - 3s 313ms/step - loss: 4.8847 - mae:
5.3677
Epoch 3/50
10/10 [==============================] - 3s 321ms/step - loss: 3.0884 - mae:
3.5605
Epoch 4/50
10/10 [==============================] - 3s 312ms/step - loss: 2.7214 - mae:
3.1892
Epoch 5/50
10/10 [==============================] - 3s 314ms/step - loss: 2.5054 - mae:
2.9683
Epoch 6/50
10/10 [==============================] - 3s 312ms/step - loss: 2.2066 - mae:
2.6635
Epoch 7/50
10/10 [==============================] - 3s 314ms/step - loss: 2.0183 - mae:
2.4761
Epoch 8/50
10/10 [==============================] - 3s 313ms/step - loss: 2.1923 - mae:
2.6506
Epoch 9/50
10/10 [==============================] - 3s 312ms/step - loss: 1.9554 - mae:
2.4061
Epoch 10/50
10/10 [==============================] - 3s 322ms/step - loss: 1.8985 - mae:
2.3529
Epoch 11/50
10/10 [==============================] - 3s 321ms/step - loss: 1.8650 - mae:
```

```
2.3167
Epoch 12/50
10/10 [==============================] - 3s 320ms/step - loss: 1.8076 - mae:
2.2599
Epoch 13/50
10/10 [==============================] - 3s 322ms/step - loss: 1.7318 - mae:
2.1847
Epoch 14/50
10/10 [==============================] - 3s 313ms/step - loss: 1.6782 - mae:
2.1239
Epoch 15/50
10/10 [==============================] - 3s 300ms/step - loss: 1.6421 - mae:
2.0915
Epoch 16/50
10/10 [==============================] - 3s 298ms/step - loss: 1.6392 - mae:
2.0862
Epoch 17/50
10/10 [==============================] - 3s 290ms/step - loss: 1.6328 - mae:
2.0789
Epoch 18/50
10/10 [==============================] - 3s 291ms/step - loss: 1.5889 - mae:
2.0325
Epoch 19/50
10/10 [==============================] - 3s 290ms/step - loss: 1.5938 - mae:
2.0387
Epoch 20/50
10/10 [==============================] - 3s 292ms/step - loss: 1.5731 - mae:
2.0160
Epoch 21/50
10/10 [==============================] - 3s 281ms/step - loss: 1.6166 - mae:
2.0641
Epoch 22/50
10/10 [==============================] - 3s 288ms/step - loss: 1.5587 - mae:
2.0057
Epoch 23/50
10/10 [==============================] - 3s 289ms/step - loss: 1.5235 - mae:
1.9648
Epoch 24/50
10/10 [==============================] - 3s 282ms/step - loss: 1.5783 - mae:
2.0272
Epoch 25/50
10/10 [==============================] - 3s 291ms/step - loss: 1.5424 - mae:
1.9862
Epoch 26/50
10/10 [==============================] - 3s 289ms/step - loss: 1.5053 - mae:
1.9462
Epoch 27/50
10/10 [==============================] - 3s 289ms/step - loss: 1.5159 - mae:
```

```
1.9554
Epoch 28/50
10/10 [==============================] - 3s 290ms/step - loss: 1.4968 - mae:
1.9377
Epoch 29/50
10/10 [==============================] - 3s 291ms/step - loss: 1.5693 - mae:
2.0150
Epoch 30/50
10/10 [==============================] - 3s 299ms/step - loss: 1.5265 - mae:
1.9660
Epoch 31/50
10/10 [==============================] - 3s 289ms/step - loss: 1.6048 - mae:
2.0485
Epoch 32/50
10/10 [==============================] - 3s 289ms/step - loss: 1.4979 - mae:
1.9383
Epoch 33/50
10/10 [==============================] - 3s 289ms/step - loss: 1.4938 - mae:
1.9356
Epoch 34/50
10/10 [==============================] - 3s 290ms/step - loss: 1.4975 - mae:
1.9365
Epoch 35/50
10/10 [==============================] - 3s 289ms/step - loss: 1.4805 - mae:
1.9211
Epoch 36/50
10/10 [==============================] - 3s 289ms/step - loss: 1.5131 - mae:
1.9528
Epoch 37/50
10/10 [==============================] - 3s 290ms/step - loss: 1.4938 - mae:
1.9343
Epoch 38/50
10/10 [==============================] - 3s 291ms/step - loss: 1.4972 - mae:
1.9376
Epoch 39/50
10/10 [==============================] - 3s 290ms/step - loss: 1.4945 - mae:
1.9337
Epoch 40/50
10/10 [==============================] - 3s 289ms/step - loss: 1.5174 - mae:
1.9599
Epoch 41/50
10/10 [==============================] - 3s 290ms/step - loss: 1.4840 - mae:
1.9237
Epoch 42/50
10/10 [==============================] - 3s 299ms/step - loss: 1.4756 - mae:
1.9152
Epoch 43/50
10/10 [==============================] - 3s 291ms/step - loss: 1.4737 - mae:
```

```
1.9120
Epoch 44/50
10/10 [==============================] - 3s 291ms/step - loss: 1.5190 - mae:
1.9602
Epoch 45/50
10/10 [==============================] - 3s 299ms/step - loss: 1.5000 - mae:
1.9416
Epoch 46/50
10/10 [==============================] - 3s 292ms/step - loss: 1.5684 - mae:
2.0134
Epoch 47/50
10/10 [==============================] - 3s 298ms/step - loss: 1.5341 - mae:
1.9772
Epoch 48/50
10/10 [==============================] - 3s 288ms/step - loss: 1.4854 - mae:
1.9279
Epoch 49/50
10/10 [==============================] - 3s 289ms/step - loss: 1.4854 - mae:
1.9271
Epoch 50/50
10/10 [==============================] - 3s 281ms/step - loss: 1.4712 - mae:
1.9107
```

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:

[15]:
```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 saw a faster approach compared to using a for loop to compute the forecasts for every point in the sequence. Remember that this faster approach uses 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

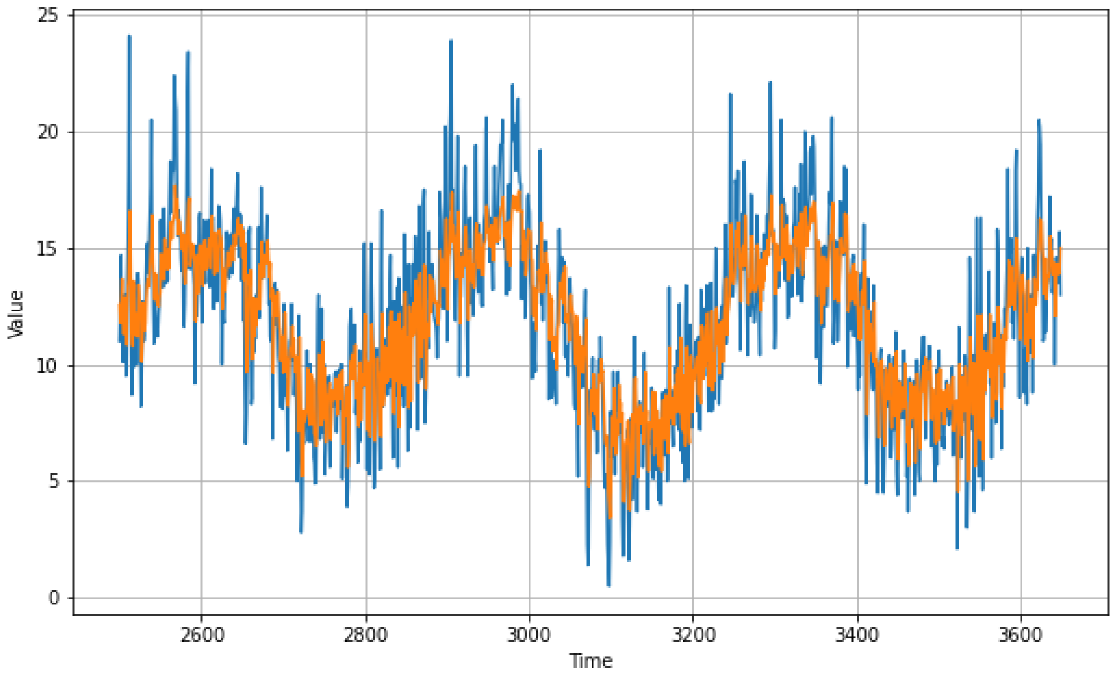
[16]:
```
def model_forecast(model, series, window_size):
    ds = tf.data.Dataset.from_tensor_slices(series)
    ds = ds.window(window_size, shift=1, drop_remainder=True)
    ds = ds.flat_map(lambda w: w.batch(window_size))
    ds = ds.batch(32).prefetch(1)
    forecast = model.predict(ds)
    return forecast
```

Now compute the actual forecast:

Note: Don't modify the cell below.

The grader uses the same slicing to get the forecast so if you change the cell below you risk having issues when submitting your model for grading.

[17]:
```
# 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 the forecast
plt.figure(figsize=(10, 6))
plot_series(time_valid, series_valid)
plot_series(time_valid, rnn_forecast)
```



[18]:
```
mse, mae = compute_metrics(series_valid, rnn_forecast)

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

```
mse: 5.31, mae: 1.80 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 this threshold run the following cell to save the model in a HDF5 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.**

[19]:
```
# 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/
```

```
WARNING:absl:Found untraced functions such as lstm_cell_4_layer_call_fn,
lstm_cell_4_layer_call_and_return_conditional_losses, lstm_cell_5_layer_call_fn,
lstm_cell_5_layer_call_and_return_conditional_losses, lstm_cell_4_layer_call_fn
while saving (showing 5 of 10). These functions will not be directly callable
after loading.

INFO:tensorflow:Assets written to: saved_model/my_model/assets

INFO:tensorflow:Assets written to: saved_model/my_model/assets
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7ff12837dd90> has the
same name 'LSTMCell' as a built-in Keras object. Consider renaming <class
'keras.layers.recurrent.LSTMCell'> to avoid naming conflicts when loading with
`tf.keras.models.load_model`. If renaming is not possible, pass the object in
the `custom_objects` parameter of the load function.
WARNING:absl:<keras.layers.recurrent.LSTMCell object at 0x7ff1283076a0> has the
same name 'LSTMCell' as a built-in Keras object. Consider renaming <class
'keras.layers.recurrent.LSTMCell'> to avoid naming conflicts when loading with
`tf.keras.models.load_model`. If renaming is not possible, pass the object in
the `custom_objects` parameter of the load function.

saved_model/
saved_model/my_model/
saved_model/my_model/keras_metadata.pb
saved_model/my_model/variables/
saved_model/my_model/variables/variables.data-00000-of-00001
```

```
saved_model/my_model/variables/variables.index
saved_model/my_model/saved_model.pb
saved_model/my_model/assets/
```

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!!!