Deep learning for timeseries

Weather Forecasting Using Time Series

In [1]:
```
!wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
!unzip jena_climate_2009_2016.csv.zip
```

```
--2024-04-07 15:20:13--  https://s3.amazonaws.com/keras-datasets/jena_climate_2009_20
16.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.217.8.14, 52.217.234.152, 52.216.
78.94, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.217.8.14|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13565642 (13M) [application/zip]
Saving to: ‘jena_climate_2009_2016.csv.zip’

jena_climate_2009_2 100%[===================>]  12.94M  14.6MB/s    in 0.9s

2024-04-07 15:20:14 (14.6 MB/s) - ‘jena_climate_2009_2016.csv.zip’ saved [13565642/13
565642]

Archive:  jena_climate_2009_2016.csv.zip
  inflating: jena_climate_2009_2016.csv
  inflating: __MACOSX/._jena_climate_2009_2016.csv
```

Inspecting the data of the Jena weather dataset

In [2]:
```
import os
fname = os.path.join("jena_climate_2009_2016.csv")

with open(fname) as f:
    data = f.read()

lines = data.split("\n")
header = lines[0].split(",")
lines = lines[1:]
print(header)
print(len(lines))
```

```
['"Date Time"', '"p (mbar)"', '"T (degC)"', '"Tpot (K)"', '"Tdew (degC)"', '"rh
(%)"', '"VPmax (mbar)"', '"VPact (mbar)"', '"VPdef (mbar)"', '"sh (g/kg)"', '"H2OC (m
mol/mol)"', '"rho (g/m**3)"', '"wv (m/s)"', '"max. wv (m/s)"', '"wd (deg)"']
420451
```

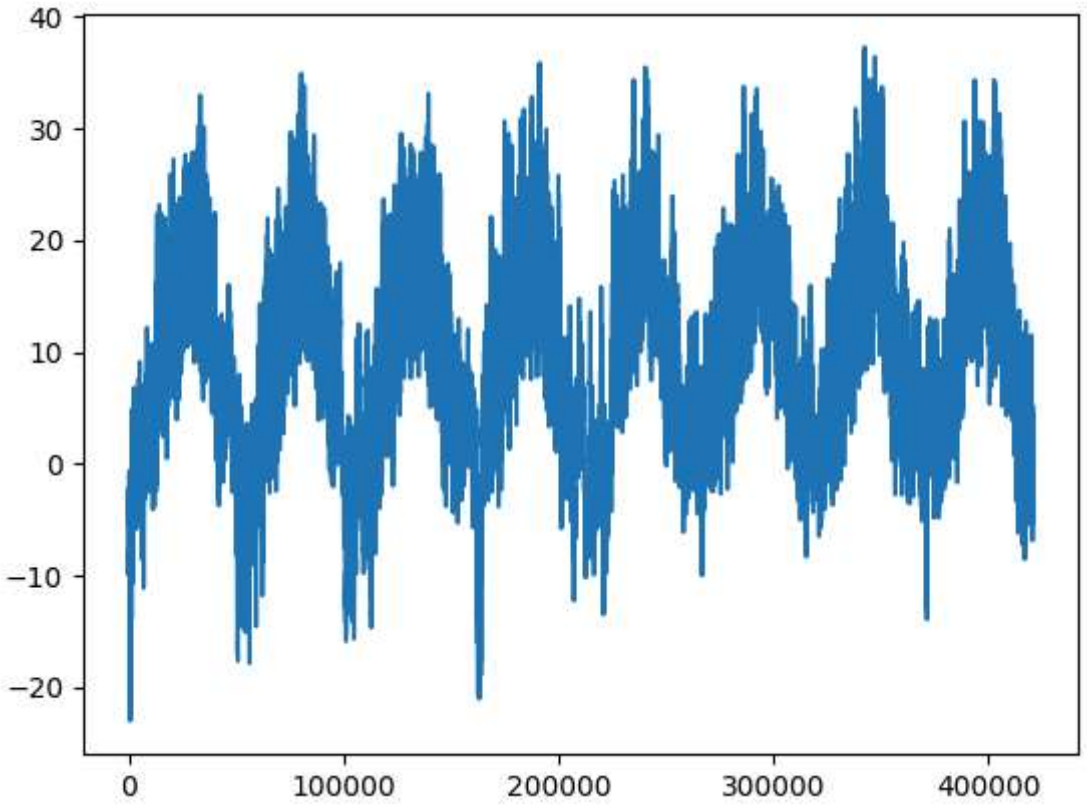
Parsing the data

In [3]:
```
import numpy as np
temperature = np.zeros((len(lines),))
raw_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(",")[1:]]
    temperature[i] = values[1]
    raw_data[i, :] = values[:]
```

Plotting the temperature timeseries

In [4]:
```
from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)
```

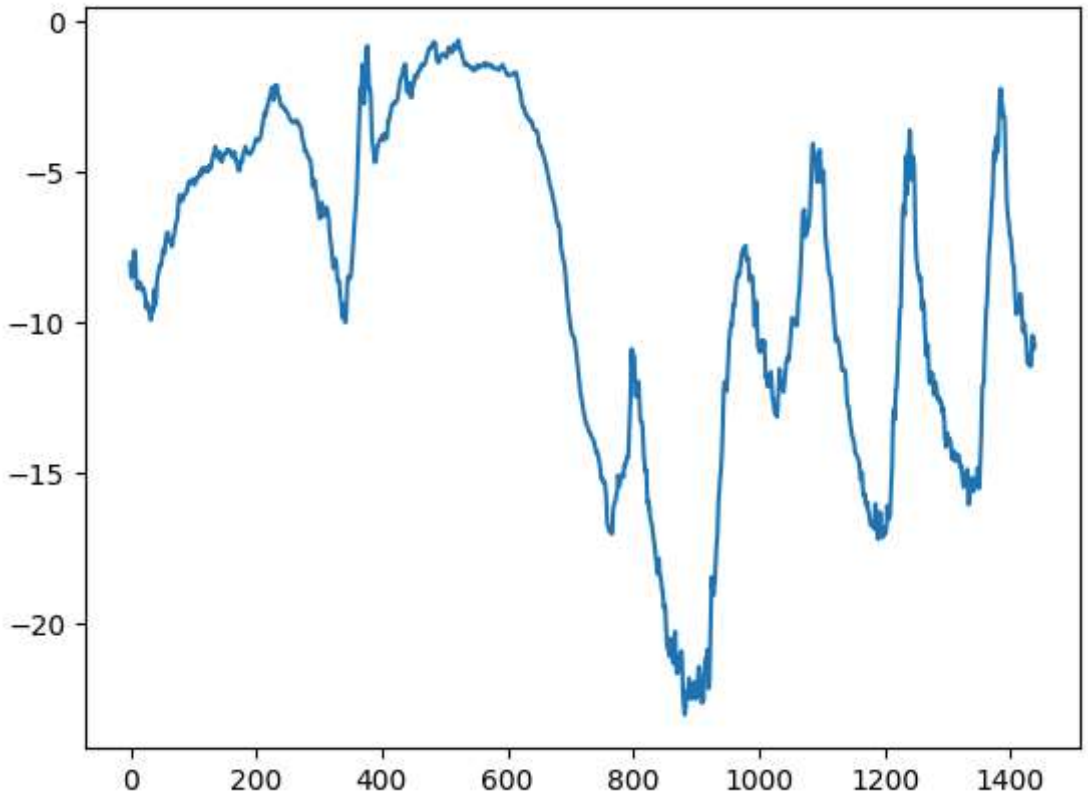
Out[4]:
```
[<matplotlib.lines.Line2D at 0x7b3c253d2d70>]
```



Plotting the first 10 days of the temperature timeseries

In [5]:
```
plt.plot(range(1440), temperature[:1440])
```

Out[5]:
```
[<matplotlib.lines.Line2D at 0x7b3c1a71ba90>]
```



Computing the number of samples we'll use for each data split

In [6]:
```
num_train_samples = int(0.5 * len(raw_data))
num_val_samples = int(0.25 * len(raw_data))
num_test_samples = len(raw_data) - num_train_samples - num_val_samples
print("num_train_samples:", num_train_samples)
print("num_val_samples:", num_val_samples)
print("num_test_samples:", num_test_samples)
```

```
num_train_samples: 210225
num_val_samples: 105112
num_test_samples: 105114
```

Preparing the data

Normalizing the data

In [7]:
```
mean = raw_data[:num_train_samples].mean(axis=0)
raw_data -= mean
std = raw_data[:num_train_samples].std(axis=0)
raw_data /= std
```

In [8]:
```
import numpy as np
from tensorflow import keras
int_sequence = np.arange(10)
dummy_dataset = keras.utils.timeseries_dataset_from_array(
    data=int_sequence[:-3],
    targets=int_sequence[3:],
    sequence_length=3,
    batch_size=2,
)
```

```
for inputs, targets in dummy_dataset:
    for i in range(inputs.shape[0]):
        print([int(x) for x in inputs[i]], int(targets[i]))
```

```
[0, 1, 2] 3
[1, 2, 3] 4
[2, 3, 4] 5
[3, 4, 5] 6
[4, 5, 6] 7
```

Instantiating datasets for training, validation, and testing

In [9]:
```
sampling_rate = 6
sequence_length = 120
delay = sampling_rate * (sequence_length + 24 - 1)
batch_size = 256

train_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=0,
    end_index=num_train_samples)

val_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples,
    end_index=num_train_samples + num_val_samples)

test_dataset = keras.utils.timeseries_dataset_from_array(
    raw_data[:-delay],
    targets=temperature[delay:],
    sampling_rate=sampling_rate,
    sequence_length=sequence_length,
    shuffle=True,
    batch_size=batch_size,
    start_index=num_train_samples + num_val_samples)
```

Inspecting the output of one of our datasets

In [10]:
```
for samples, targets in train_dataset:
    print("samples shape:", samples.shape)
    print("targets shape:", targets.shape)
    break
```

```
samples shape: (256, 120, 14)
targets shape: (256,)
```

A common-sense, non-machine-learning baseline

Computing the common-sense baseline MAE

In [11]:
```
def evaluate_naive_method(dataset):
    total_abs_err = 0.
    samples_seen = 0
    for samples, targets in dataset:
        preds = samples[:, -1, 1] * std[1] + mean[1]
        total_abs_err += np.sum(np.abs(preds - targets))
        samples_seen += samples.shape[0]
    return total_abs_err / samples_seen

print(f"Validation MAE: {evaluate_naive_method(val_dataset):.2f}")
print(f"Test MAE: {evaluate_naive_method(test_dataset):.2f}")
```

```
Validation MAE: 2.44
Test MAE: 2.62
```

A Basic model with regular calculation has been performed and the validation and test MAE is as follows:

Validation MAE: 2.44 Test MAE: 2.62

Initial Learning Model

Training and evaluating a densely connected model

- With two dense layers and 32 units in input layer with relu activation function.
- RMSprop optimizer is chosen for training the model, offering adaptive learning rates.
- Mean Squared Error (MSE) is specified as the loss function, measuring the difference between predicted and actual values.
- Mean Absolute Error (MAE) is defined as a metric to monitor during training, providing insight into the model's performance on the validation set.

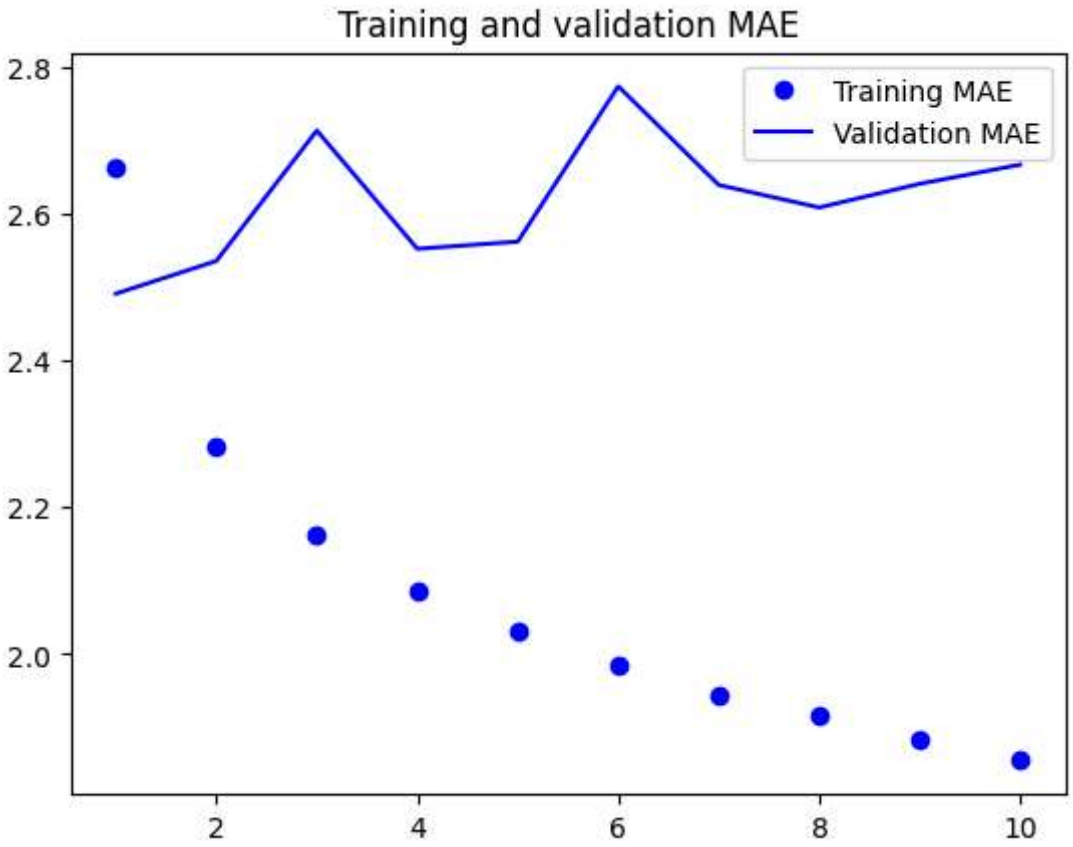
In [12]:
```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Flatten()(inputs)
x = layers.Dense(32, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 14s 16ms/step - loss: 11.9010 - mae: 2.662
9 - val_loss: 9.9414 - val_mae: 2.4914
Epoch 2/10
819/819 [==============================] - 11s 14ms/step - loss: 8.4627 - mae: 2.2831
- val_loss: 10.2159 - val_mae: 2.5361
Epoch 3/10
819/819 [==============================] - 12s 15ms/step - loss: 7.5941 - mae: 2.1638
- val_loss: 11.8024 - val_mae: 2.7140
Epoch 4/10
819/819 [==============================] - 12s 14ms/step - loss: 7.0332 - mae: 2.0845
- val_loss: 10.3962 - val_mae: 2.5526
Epoch 5/10
819/819 [==============================] - 12s 15ms/step - loss: 6.6609 - mae: 2.0307
- val_loss: 10.4689 - val_mae: 2.5625
Epoch 6/10
819/819 [==============================] - 13s 15ms/step - loss: 6.3463 - mae: 1.9840
- val_loss: 12.3527 - val_mae: 2.7743
Epoch 7/10
819/819 [==============================] - 12s 15ms/step - loss: 6.0997 - mae: 1.9438
- val_loss: 11.1778 - val_mae: 2.6397
Epoch 8/10
819/819 [==============================] - 12s 14ms/step - loss: 5.8985 - mae: 1.9149
- val_loss: 10.9268 - val_mae: 2.6090
Epoch 9/10
819/819 [==============================] - 12s 14ms/step - loss: 5.7168 - mae: 1.8835
- val_loss: 11.2303 - val_mae: 2.6413
Epoch 10/10
819/819 [==============================] - 11s 14ms/step - loss: 5.5451 - mae: 1.8546
- val_loss: 11.4959 - val_mae: 2.6675
405/405 [==============================] - 4s 9ms/step - loss: 10.9297 - mae: 2.5893
Test MAE: 2.59
```

Obtained a test MAE of **2.62** with densely connected model

Plotting results

In [13]:
```python
import matplotlib.pyplot as plt
loss = history.history["mae"]
val_loss = history.history["val_mae"]
epochs = range(1, len(loss) + 1)
plt.figure()
plt.plot(epochs, loss, "bo", label="Training MAE")
plt.plot(epochs, val_loss, "b", label="Validation MAE")
plt.title("Training and validation MAE")
plt.legend()
plt.show()
```



Let's try a 1D convolutional model

In [14]:
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 6, activation="relu")(x)
x = layers.GlobalAveragePooling1D()(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_conv.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_conv.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 15s 15ms/step - loss: 25.5756 - mae: 3.907
4 - val_loss: 16.2414 - val_mae: 3.1606
Epoch 2/10
819/819 [==============================] - 13s 15ms/step - loss: 15.7580 - mae: 3.151
4 - val_loss: 15.7541 - val_mae: 3.0715
Epoch 3/10
819/819 [==============================] - 12s 15ms/step - loss: 14.3575 - mae: 3.007
2 - val_loss: 14.6128 - val_mae: 2.9698
Epoch 4/10
819/819 [==============================] - 12s 15ms/step - loss: 13.4928 - mae: 2.911
7 - val_loss: 15.3138 - val_mae: 3.0876
Epoch 5/10
819/819 [==============================] - 12s 14ms/step - loss: 12.7442 - mae: 2.823
2 - val_loss: 15.9253 - val_mae: 3.1454
Epoch 6/10
819/819 [==============================] - 12s 15ms/step - loss: 12.1946 - mae: 2.758
9 - val_loss: 14.0912 - val_mae: 2.9465
Epoch 7/10
819/819 [==============================] - 12s 14ms/step - loss: 11.6940 - mae: 2.701
8 - val_loss: 14.2415 - val_mae: 2.9575
Epoch 8/10
819/819 [==============================] - 12s 15ms/step - loss: 11.2607 - mae: 2.652
2 - val_loss: 14.6832 - val_mae: 3.0235
Epoch 9/10
819/819 [==============================] - 13s 16ms/step - loss: 10.9191 - mae: 2.612
1 - val_loss: 15.1459 - val_mae: 3.0535
Epoch 10/10
819/819 [==============================] - 12s 14ms/step - loss: 10.6071 - mae: 2.575
3 - val_loss: 15.1200 - val_mae: 3.0549
405/405 [==============================] - 4s 9ms/step - loss: 16.5499 - mae: 3.2040
Test MAE: 3.20
```

A regular 1D convultional network yielded a test MAE of 3.2 which is more than the dense layer network means it is underperforming.

A first recurrent baseline

A simple LSTM-based model

In [15]:
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(16)(inputs)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)

model = keras.models.load_model("jena_lstm.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 15s 16ms/step - loss: 45.8279 - mae: 4.946
3 - val_loss: 13.4332 - val_mae: 2.7469
Epoch 2/10
819/819 [==============================] - 13s 15ms/step - loss: 11.3122 - mae: 2.608
2 - val_loss: 9.4794 - val_mae: 2.3836
Epoch 3/10
819/819 [==============================] - 12s 15ms/step - loss: 9.8318 - mae: 2.4480
- val_loss: 9.8560 - val_mae: 2.4013
Epoch 4/10
819/819 [==============================] - 12s 15ms/step - loss: 9.3402 - mae: 2.3843
- val_loss: 10.0675 - val_mae: 2.4337
Epoch 5/10
819/819 [==============================] - 13s 15ms/step - loss: 9.0368 - mae: 2.3434
- val_loss: 10.2675 - val_mae: 2.4506
Epoch 6/10
819/819 [==============================] - 12s 15ms/step - loss: 8.7933 - mae: 2.3098
- val_loss: 10.8339 - val_mae: 2.4908
Epoch 7/10
819/819 [==============================] - 12s 15ms/step - loss: 8.5939 - mae: 2.2814
- val_loss: 10.7766 - val_mae: 2.4741
Epoch 8/10
819/819 [==============================] - 12s 15ms/step - loss: 8.4353 - mae: 2.2602
- val_loss: 11.1890 - val_mae: 2.5243
Epoch 9/10
819/819 [==============================] - 12s 15ms/step - loss: 8.2766 - mae: 2.2401
- val_loss: 11.2528 - val_mae: 2.5308
Epoch 10/10
819/819 [==============================] - 12s 15ms/step - loss: 8.1471 - mae: 2.2219
- val_loss: 11.4397 - val_mae: 2.5543
405/405 [==============================] - 5s 9ms/step - loss: 10.8730 - mae: 2.5908
Test MAE: 2.59
```

A basic baseline RNN was built using LSTM and the test MAE has improved to 2.59

Understanding recurrent neural networks

NumPy implementation of a simple RNN

In [16]:
```
import numpy as np
timesteps = 100
input_features = 32
output_features = 64
inputs = np.random.random((timesteps, input_features))
state_t = np.zeros((output_features,))
W = np.random.random((output_features, input_features))
U = np.random.random((output_features, output_features))
b = np.random.random((output_features,))
successive_outputs = []
for input_t in inputs:
    output_t = np.tanh(np.dot(W, input_t) + np.dot(U, state_t) + b)
    successive_outputs.append(output_t)
    state_t = output_t
final_output_sequence = np.stack(successive_outputs, axis=0)
```

1. Adjusting the number of units in each recurrent layer in the stacked setup

Using SimpleRNN in Keras

Stacking RNN layers

- Stacked SimpleRNN layers with increasing units (32, 32) process sequential data.
- RMSprop optimizer is used with Mean Squared Error (MSE) loss and Mean Absolute Error (MAE) metric.

In [26]:
```
steps = 120
num_features = 32
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(32, return_sequences=True)(inputs)
x = layers.SimpleRNN(32, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_simple_rnn.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

```
Epoch 1/10
819/819 [==============================] - 261s 315ms/step - loss: 135.8822 - mae: 9.
5040 - val_loss: 143.3909 - val_mae: 9.8324
Epoch 2/10
819/819 [==============================] - 253s 309ms/step - loss: 135.8361 - mae: 9.
4966 - val_loss: 143.4218 - val_mae: 9.8387
Epoch 3/10
819/819 [==============================] - 252s 308ms/step - loss: 135.8122 - mae: 9.
4922 - val_loss: 143.3909 - val_mae: 9.8320
Epoch 4/10
819/819 [==============================] - 253s 309ms/step - loss: 135.7956 - mae: 9.
4891 - val_loss: 143.4257 - val_mae: 9.8390
Epoch 5/10
819/819 [==============================] - 252s 308ms/step - loss: 135.7781 - mae: 9.
4858 - val_loss: 143.4148 - val_mae: 9.8374
Epoch 6/10
819/819 [==============================] - 252s 307ms/step - loss: 135.7660 - mae: 9.
4837 - val_loss: 143.4137 - val_mae: 9.8388
Epoch 7/10
819/819 [==============================] - 252s 307ms/step - loss: 135.7719 - mae: 9.
4829 - val_loss: 143.4086 - val_mae: 9.8356
Epoch 8/10
819/819 [==============================] - 249s 304ms/step - loss: 135.7633 - mae: 9.
4819 - val_loss: 143.4271 - val_mae: 9.8398
Epoch 9/10
819/819 [==============================] - 248s 303ms/step - loss: 135.7498 - mae: 9.
4801 - val_loss: 143.4514 - val_mae: 9.8432
Epoch 10/10
819/819 [==============================] - 249s 304ms/step - loss: 135.7297 - mae: 9.
4766 - val_loss: 143.4532 - val_mae: 9.8425
```

In [36]:
```
model = keras.models.load_model("jena_simple_rnn.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
405/405 [==============================] - 18s 41ms/step - loss: 151.1017 - mae: 9.90
21
Test MAE: 9.90
```

- A simpleRNN with two layer has a MAE of 9.9
- The error is very large when compared to simple lstm model

2. Using layer_lstm() instead of layer_gru()

Stacking RNNs with GRU and LSTM

Training and evaluating a dropout-regularized, stacked GRU model

- Two stacked GRU layers are employed, with 64 units in the first layer and 32 units in the second layer.
- The second GRU layer is followed by a dropout layer with a dropout rate of 0.4 to prevent overfitting.
- The model is compiled using the RMSprop optimizer, Mean Squared Error (MSE) loss function, and Mean Absolute Error (MAE) metric.

In [27]:
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(64, return_sequences=True)(inputs)
x = layers.GRU(32)(x)
x = layers.Dropout(0.4)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_stacked_gru_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)
model = keras.models.load_model("jena_stacked_gru_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 20s 20ms/step - loss: 20.0742 - mae: 3.324
3 - val_loss: 9.6428 - val_mae: 2.4327
Epoch 2/10
819/819 [==============================] - 16s 20ms/step - loss: 11.8389 - mae: 2.677
0 - val_loss: 9.0845 - val_mae: 2.3585
Epoch 3/10
819/819 [==============================] - 16s 20ms/step - loss: 10.5393 - mae: 2.522
7 - val_loss: 9.3764 - val_mae: 2.3874
Epoch 4/10
819/819 [==============================] - 16s 19ms/step - loss: 9.2671 - mae: 2.3674
- val_loss: 10.6624 - val_mae: 2.5509
Epoch 5/10
819/819 [==============================] - 16s 19ms/step - loss: 8.0671 - mae: 2.2013
- val_loss: 11.3387 - val_mae: 2.6359
Epoch 6/10
819/819 [==============================] - 16s 19ms/step - loss: 7.0353 - mae: 2.0519
- val_loss: 12.7128 - val_mae: 2.7802
Epoch 7/10
819/819 [==============================] - 16s 19ms/step - loss: 6.2748 - mae: 1.9318
- val_loss: 12.5647 - val_mae: 2.7348
Epoch 8/10
819/819 [==============================] - 16s 19ms/step - loss: 5.6409 - mae: 1.8277
- val_loss: 12.6389 - val_mae: 2.7423
Epoch 9/10
819/819 [==============================] - 16s 20ms/step - loss: 5.2274 - mae: 1.7510
- val_loss: 13.2001 - val_mae: 2.8214
Epoch 10/10
819/819 [==============================] - 16s 19ms/step - loss: 4.8644 - mae: 1.6852
- val_loss: 12.7056 - val_mae: 2.7433
405/405 [==============================] - 5s 10ms/step - loss: 10.0510 - mae: 2.4652
Test MAE: 2.47
```

- Using GRU stacked RNN the test MAE reduced to even more to **2.47**.
- It can be seen that a stacked two layer GRU RNN has better results than simpleRNN

Training and evaluating a dropout-regularized LSTM

- This model comprises two LSTM (Long Short-Term Memory) layers. The first layer has 64 units, followed by a second layer with 32 units.
- A dropout layer with a dropout rate of 0.4 is inserted between the two LSTM layers. Dropout is effective for regularizing the model and reducing overfitting by randomly dropping 40% of the units during training.
- The model is compiled using the RMSprop optimizer, a robust optimizer for training recurrent neural networks.
- Mean Squared Error (MSE) is chosen as the loss function to measure the difference between predicted and actual values.
- Mean Absolute Error (MAE) is selected as a metric to monitor during training, providing insight into the model's performance on the validation set.

In [32]:
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(64, return_sequences=True)(inputs)
x = layers.LSTM(32)(x)
x = layers.Dropout(0.4)(x)
```

```
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm_dropout.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)
model = keras.models.load_model("jena_lstm_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Epoch 1/10
819/819 [==============================] - 20s 21ms/step - loss: 22.4971 - mae: 3.471
4 - val_loss: 9.8105 - val_mae: 2.4364
Epoch 2/10
819/819 [==============================] - 16s 19ms/step - loss: 10.8545 - mae: 2.556
8 - val_loss: 10.2882 - val_mae: 2.5084
Epoch 3/10
819/819 [==============================] - 16s 19ms/step - loss: 8.9814 - mae: 2.3074
- val_loss: 11.6340 - val_mae: 2.6451
Epoch 4/10
819/819 [==============================] - 16s 20ms/step - loss: 7.5244 - mae: 2.0975
- val_loss: 12.6035 - val_mae: 2.7821
Epoch 5/10
819/819 [==============================] - 16s 19ms/step - loss: 6.4987 - mae: 1.9364
- val_loss: 12.7847 - val_mae: 2.8004
Epoch 6/10
819/819 [==============================] - 16s 19ms/step - loss: 5.7661 - mae: 1.8157
- val_loss: 12.6321 - val_mae: 2.7941
Epoch 7/10
819/819 [==============================] - 16s 20ms/step - loss: 5.1777 - mae: 1.7142
- val_loss: 12.9884 - val_mae: 2.8087
Epoch 8/10
819/819 [==============================] - 16s 19ms/step - loss: 4.7558 - mae: 1.6349
- val_loss: 12.6791 - val_mae: 2.7959
Epoch 9/10
819/819 [==============================] - 16s 19ms/step - loss: 4.4257 - mae: 1.5734
- val_loss: 12.5862 - val_mae: 2.7619
Epoch 10/10
819/819 [==============================] - 16s 19ms/step - loss: 4.1716 - mae: 1.5207
- val_loss: 13.5707 - val_mae: 2.8673
405/405 [==============================] - 5s 10ms/step - loss: 11.0071 - mae: 2.6070
Test MAE: 2.61
```

- With LSTM, the test MAE is 2.61 which is little similar to GRU.
- Both LSTM and GRU performed similarly with slight changes.

Using bidirectional RNNs

Training and evaluating a bidirectional LSTM

In [29]:
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Bidirectional(layers.LSTM(16))(inputs)
```

```
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset)
```

```
Epoch 1/10
819/819 [==============================] - 19s 19ms/step - loss: 26.4161 - mae: 3.704
2 - val_loss: 10.7981 - val_mae: 2.5559
Epoch 2/10
819/819 [==============================] - 15s 18ms/step - loss: 9.5371 - mae: 2.4142
- val_loss: 9.8833 - val_mae: 2.4341
Epoch 3/10
819/819 [==============================] - 15s 18ms/step - loss: 8.5709 - mae: 2.2838
- val_loss: 9.8581 - val_mae: 2.4395
Epoch 4/10
819/819 [==============================] - 15s 19ms/step - loss: 7.9988 - mae: 2.2052
- val_loss: 9.7590 - val_mae: 2.4161
Epoch 5/10
819/819 [==============================] - 15s 18ms/step - loss: 7.4943 - mae: 2.1336
- val_loss: 9.7918 - val_mae: 2.4288
Epoch 6/10
819/819 [==============================] - 15s 18ms/step - loss: 7.0928 - mae: 2.0778
- val_loss: 9.9425 - val_mae: 2.4526
Epoch 7/10
819/819 [==============================] - 15s 18ms/step - loss: 6.8505 - mae: 2.0421
- val_loss: 10.5840 - val_mae: 2.5636
Epoch 8/10
819/819 [==============================] - 15s 19ms/step - loss: 6.5960 - mae: 2.0052
- val_loss: 10.0918 - val_mae: 2.4703
Epoch 9/10
819/819 [==============================] - 15s 18ms/step - loss: 6.4019 - mae: 1.9734
- val_loss: 10.7107 - val_mae: 2.5517
Epoch 10/10
819/819 [==============================] - 15s 18ms/step - loss: 6.2415 - mae: 1.9500
- val_loss: 10.6019 - val_mae: 2.5310
```

3. Using a combination of 1d_convnets and RNN.

- A conv 1D stacked with RNN LSTM

In [33]:
```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.Conv1D(8, 24, activation="relu")(inputs)
x = layers.MaxPooling1D(2)(x)
x = layers.Conv1D(8, 12, activation="relu")(x)
x = layers.MaxPooling1D(2)(x)
x = layers.LSTM(32)(x)
x = layers.Dropout(0.6)(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
    keras.callbacks.ModelCheckpoint("jena_lstm_conv_dropout.keras",
```

```
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

```
Epoch 1/10
819/819 [==============================] - 16s 16ms/step - loss: 31.6672 - mae: 4.201
3 - val_loss: 12.6290 - val_mae: 2.7577
Epoch 2/10
819/819 [==============================] - 13s 16ms/step - loss: 18.0107 - mae: 3.289
1 - val_loss: 13.6961 - val_mae: 2.9287
Epoch 3/10
819/819 [==============================] - 13s 15ms/step - loss: 16.3854 - mae: 3.133
4 - val_loss: 11.4190 - val_mae: 2.6671
Epoch 4/10
819/819 [==============================] - 13s 16ms/step - loss: 15.4951 - mae: 3.041
0 - val_loss: 12.8659 - val_mae: 2.8340
Epoch 5/10
819/819 [==============================] - 13s 15ms/step - loss: 14.7795 - mae: 2.961
5 - val_loss: 13.8016 - val_mae: 2.9606
Epoch 6/10
819/819 [==============================] - 12s 15ms/step - loss: 14.1035 - mae: 2.891
2 - val_loss: 12.5088 - val_mae: 2.8081
Epoch 7/10
819/819 [==============================] - 13s 15ms/step - loss: 13.5332 - mae: 2.825
7 - val_loss: 13.7193 - val_mae: 2.9630
Epoch 8/10
819/819 [==============================] - 13s 15ms/step - loss: 12.9905 - mae: 2.766
1 - val_loss: 13.0015 - val_mae: 2.8453
Epoch 9/10
819/819 [==============================] - 12s 15ms/step - loss: 12.6623 - mae: 2.727
6 - val_loss: 13.1087 - val_mae: 2.8756
Epoch 10/10
819/819 [==============================] - 12s 15ms/step - loss: 12.1598 - mae: 2.672
8 - val_loss: 14.5883 - val_mae: 3.0124
```

In [34]:
```
model = keras.models.load_model("jena_lstm_conv_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
405/405 [==============================] - 5s 9ms/step - loss: 13.1514 - mae: 2.8470
Test MAE: 2.85
```

With combination of conv1d and RNN lstm, the model got worsened with test MAE 2.85.

Summary

Using the Jena Climate dataset as a case study, I have developed and assessed several neural network designs for time series forecasting. The main goal is to efficiently forecast future temperature values using historical climate data. The first step is to import the dataset, which includes climatic observations for Jena, Germany, from 2009 to 2016. To obtain preliminary insights, the temperature time series is visualised and the data is carefully analysed. Most importantly, to guarantee reliable model evaluation and avoid overfitting, the dataset is divided into training, validation, and test sets. A strategy that uses common sense is used to create a

baseline for comparison. The temperature is predicted using the mean of the training data, and the mean absolute error (MAE) is produced. The effectiveness of more complex models can be evaluated using this simple baseline as a benchmark.

The information explores a number of neural network topologies, each with special advantages and disadvantages:

Densely Connected Model: A simple model with an input layer of 32 units and two dense layers. Utilised are the mean squared error (MSE) loss function and the RMSprop optimizer. This model obtains a good test MAE of 2.59 in spite of its simplicity.

1D Convolutional Model: Comprising three 1D convolutional layers and max-pooling layers, this model makes use of the capabilities of convolutional neural networks (CNNs). With a higher test MAE of 3.20, it performs worse than the densely connected model.

RNNs, or recurrent neural networks: Since time series data are sequential, many RNN topologies are investigated: a straightforward RNN model with stacked layers and increasing units that makes use of Keras' SimpleRNN layer. The model's high test MAE of 9.90 indicates its poor performance.

stacked Gated Recurrent Unit (GRU) model that guards against overfitting by using dropout regularisation. The test MAE of 2.47 achieved by this model is outstanding.

Long Short-Term Memory (LSTM) model that is stacked and incorporates dropout regularisation. Its test MAE of 2.61 indicates that it performs similarly to the GRU model.

Combination of 1D Convolution and RNN: An LSTM layer, dropout regularisation, and a 1D convolutional layer are combined to create a hybrid model that aims to take advantage of the advantages of both convolutional and recurrent layers. Nevertheless, this model performs worse than the stacked GRU and LSTM models, with a test MAE of 2.85.

Among the models tested, the stacked GRU and LSTM models show to be the best performing architectures, with the lowest test MAE. Their improved performance can be attributed to their capacity to detect long-term dependencies in the time series data as well as the regularisation dropout provides. Using the Jena Climate dataset as a useful case study, this thorough examination, in its whole, offers a methodical approach to developing and assessing several neural network designs for time series forecasting. The outcomes demonstrate how well stacked GRU and LSTM models perform in comparison to other architectures investigated in identifying complex patterns and connections in the climate data.