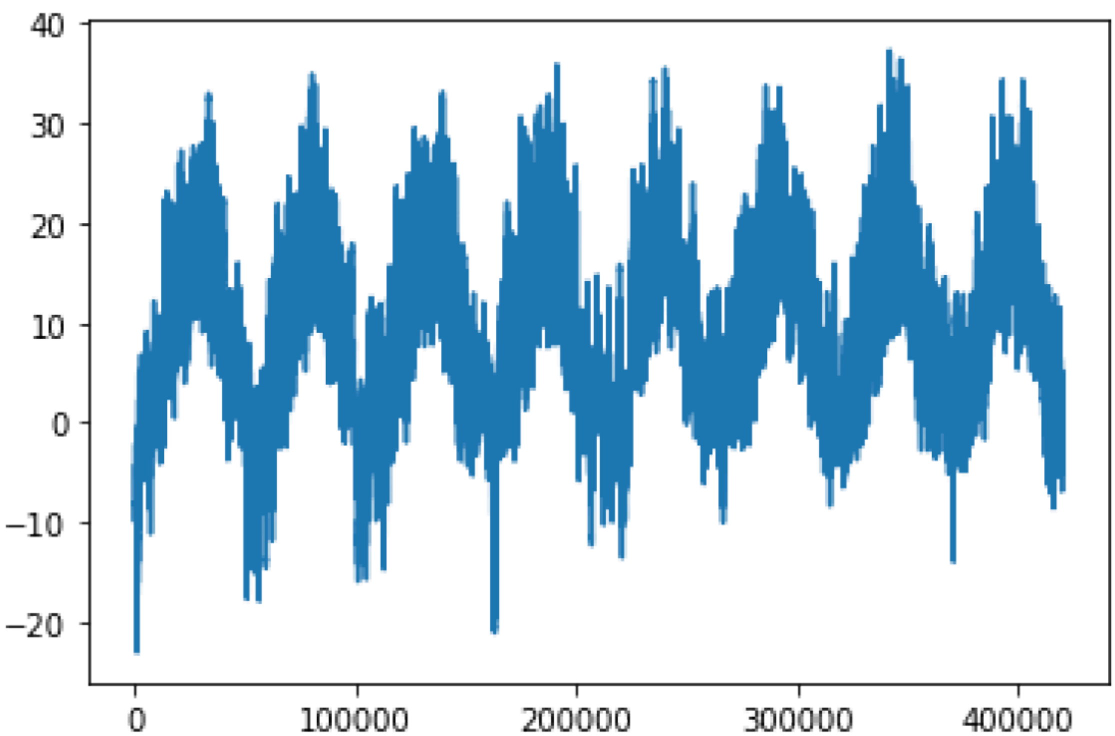
Assignment 3: Time-Series Data

Question: Use any or all of the methods we discussed in class to improve weather time-series forecasting problems discussed in class. These methods can include:

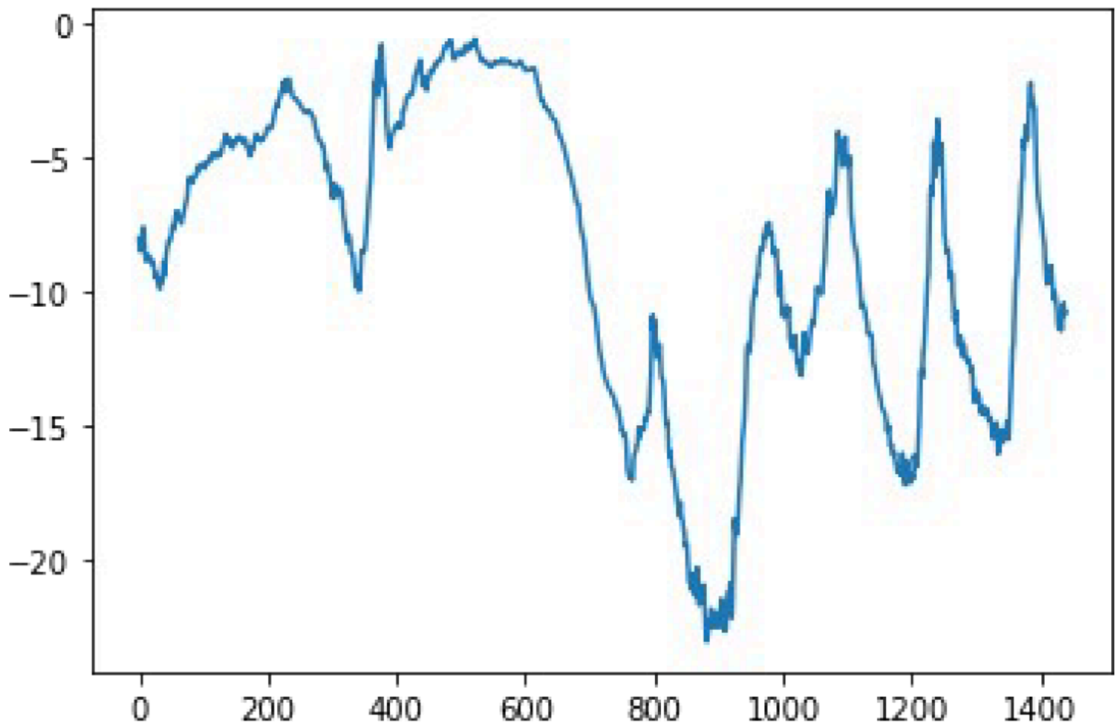
1. Adjusting the number of units in each recurrent layer in the stacked setup
2. Using layer_lstm() instead of layer_gru().
3. Using a combination of 1d_convnets and RNN.

For Time series data, densely connected network or convolutional network is not observed to be effective. So, we get introduced to RNNs (Recurrent Neural Networks). We're working with a weather timeseries dataset recorded at the weather station at the Max Planck Institute for Biogeochemistry in Jena, Germany.1 In this dataset, 14 different quantities (such as temperature, pressure, humidity, wind direction, and so on) were recorded every 10 minutes over several years. The original data goes back to 2003, but the subset of the data we've downloaded is limited to 2009–2016.

The figure below shows the plot of temperature (in degrees Celsius) over time. On this plot, we can clearly see the yearly periodicity of temperature—the data spans 8 years.



Also, the plot below shows a narrower plot of the first 10 days of temperature data. Because the data is recorded every 10 minutes, we get 24*6 = 144 data points per day. On this plot, we can see daily periodicity, especially for the last 4 days. Also note that this 10-day period must be coming from a fairly cold winter month.



We have normalized each timeseries independently so that they all take small values on a similar scale. We're going to use the first 210,225 timesteps as training data, so we'll compute the mean and standard deviation only on this fraction of the data.

Before we start using black-box deep learning models to solve the temperature prediction problem, let's try a simple, common-sense approach. It will serve as a sanity check, and it will establish a baseline that we'll have to beat in order to demonstrate the usefulness of more-advanced machine learning models. Such common-sense baselines can be useful when we're approaching a new problem for which there is no known solution (yet)

In this case, the temperature timeseries can safely be assumed to be continuous (the temperatures tomorrow are likely to be close to the temperatures today) as well as periodical with a daily period. Thus a common-sense approach is to always predict that the temperature 24 hours from now will be equal to the temperature right now.

This common-sense baseline achieves a validation MAE of 2.44 degrees Celsius and a test MAE of 2.62 degrees Celsius. So if we always assume that the temperature 24 hours in the future will be the same as it is now, we will be off by two and a half degrees on average.

As it turns out, this model performs even worse than the densely connected one, only achieving a validation MAE of about 2.9 degrees, far from the common-sense baseline. The possible reasons are: First, weather data doesn't quite respect the translation invariance assumption. While the data does feature daily cycles, data from a morning follows different properties than data from an evening or from the middle of the night. Weather data is only translation-invariant for a very specific timescale. Second, order in our data matters—a lot. The recent past is far more informative for predicting the next day's temperature than data from five days ago. A 1D convnet is not able to leverage this fact. In particular, our max pooling and global average pooling layers are largely destroying order information.

There's a family of neural network architectures designed specifically for this use case: recurrent neural networks. Among them, the Long Short-Term Memory (LSTM) layer has long been very popular. We use a simple RNN initially however, SimpleRNN has a major issue: although it should theoretically be able to retain at time t information about inputs seen many timesteps before, such long-term dependencies prove impossible to learn in practice. This is due to the vanishing gradient problem. Thankfully, SimpleRNN isn't the only recurrent layer available in Keras. There are two others, LSTM and GRU, which were designed to address these issues.Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers. Recurrent layer stacking is a classic way to build more-powerful recurrent networks: for instance, not too long ago the Google Translate algorithm was powered by a stack of seven large LSTM layers—that's huge.The last technique we'll look at in this section is the bidirectional RNN. A bidirectional RNN is a common RNN variant that can offer greater performance than a regular RNN on certain tasks. It's frequently used in natural language processing—we could call it the Swiss Army knife of deep learning for natural language processing.A bidirectional RNN exploits the order sensitivity of RNNs: it uses two regular RNNs, such as the GRU and LSTM layers we're already familiar with, each of which processes the input sequence in one direction (chronologically and antichronologically), and then merges their representations. By processing a sequence both ways, a bidirectional RNN can catch patterns that may be overlooked by a unidirectional RNN

The reversed-order LSTM strongly underperforms even the common-sense baseline, indicating that in this case, chronological processing is important to the success of the approach. This makes perfect sense: the underlying LSTM layer will typically be better at remembering the recent past than the distant past, and naturally the more recent weather data points are more predictive than older data points for the problem (that's what makes the common-sense baseline fairly strong). Thus the chronological version of the layer is bound to outperform the reversed-order version.

Model	Training MAE	Validation MAE
Baseline Model		2.44
Densely Connected Model	2.3413	2.5069
1D Convolutional Model	2.5698	2.9408
Simple LSTM-based Model	2.3386	2.4231
Dropout- Regularized LSTM Model	2.9870	2.3682
Stacked GRU Model	2.6135	2.2910
Bidirectional LSTM Model	2.2457	2.4259

From the above table, it is evident that, the most effective model amongst the ones that was trained is the Dropout-Regularized Stacked GRU Model. After fitting the data to the test data, we achieved a test MAE of 2.44, which has beaten the Baseline Model with a test MAE of 2.62.

This is a companion notebook for the book Deep Learning with Python, Second Edition. For readability, it only contains runnable code blocks and section titles, and omits everything else in the book: text paragraphs, figures, and pseudocode.

If you want to be able to follow what's going on, I recommend reading the notebook side by side with your copy of the book.

This notebook was generated for TensorFlow 2.6.

Deep learning for timeseries

Different kinds of timeseries tasks

A temperature-forecasting example

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

```
--2024-04-06 13:55:51--
https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.217.140.200,
52.217.92.38, 52.217.104.118, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|
52.217.140.200|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13565642 (13M) [application/zip]
Saving to: ‘jena_climate_2009_2016.csv.zip.1’

100%[======================================>] 13,565,642  46.7MB/s
in 0.3s

2024-04-06 13:55:52 (46.7 MB/s) - ‘jena_climate_2009_2016.csv.zip.1’
saved [13565642/13565642]

Archive:  jena_climate_2009_2016.csv.zip
replace jena_climate_2009_2016.csv? [y]es, [n]o, [A]ll, [N]one,
[r]ename:
```

Inspecting the data of the Jena weather dataset

```
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 (mmol/mol)"', '"rho (g/m**3)"', '"wv
(m/s)"', '"max. wv (m/s)"', '"wd (deg)"']
420451
```

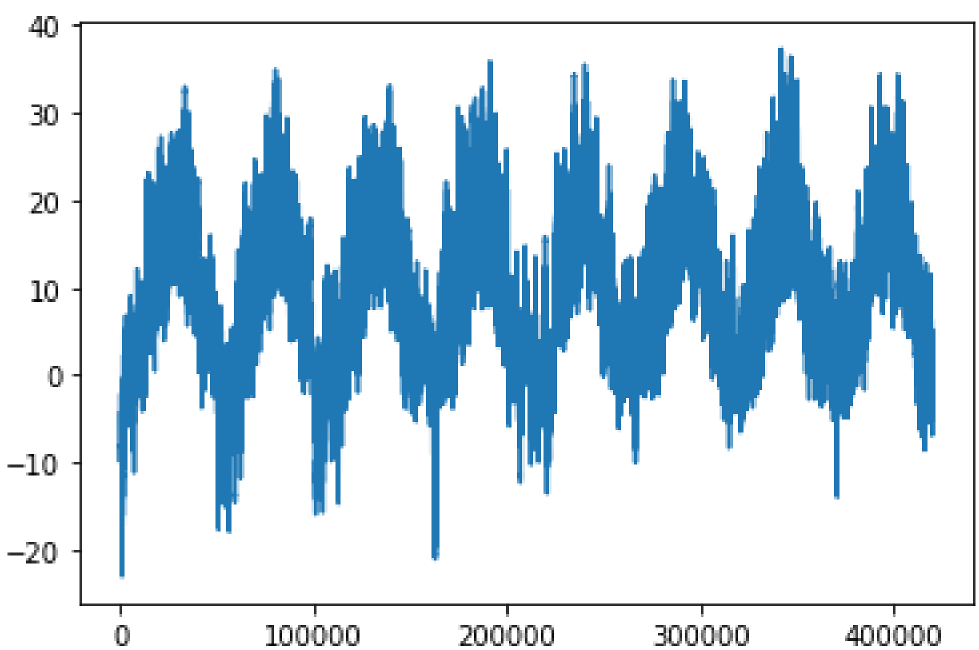
Parsing the data

```
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

```
from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)
```

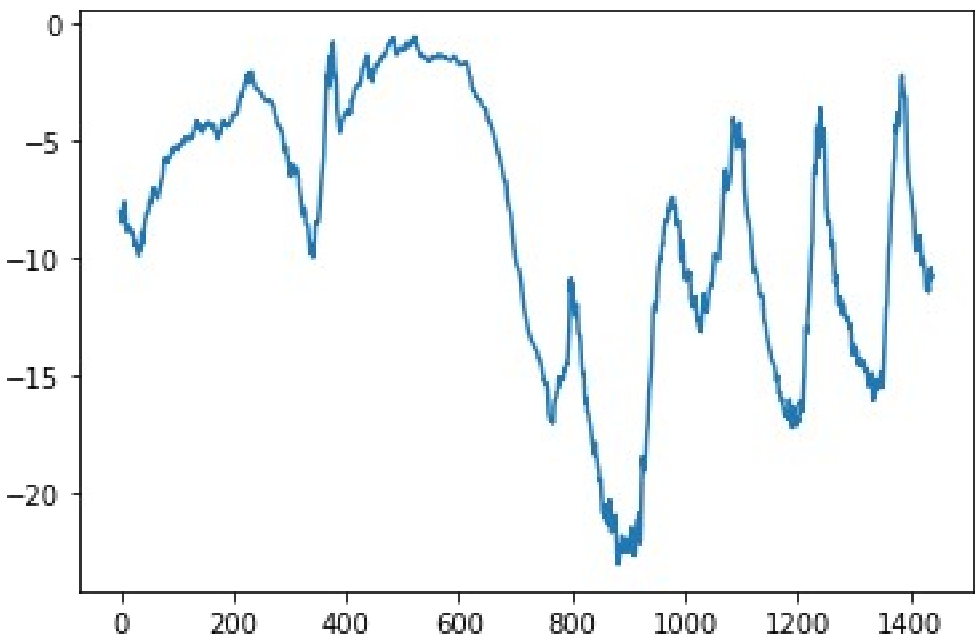
```
[<matplotlib.lines.Line2D at 0x7f3c5b658748>]
```



Plotting the first 10 days of the temperature timeseries

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

```
[<matplotlib.lines.Line2D at 0x7f3c5354ef60>]
```



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

```
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

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

```python
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

```python
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

```
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

```
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
```

Let's try a basic machine-learning model

Training and evaluating a densely connected model

```
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(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
```

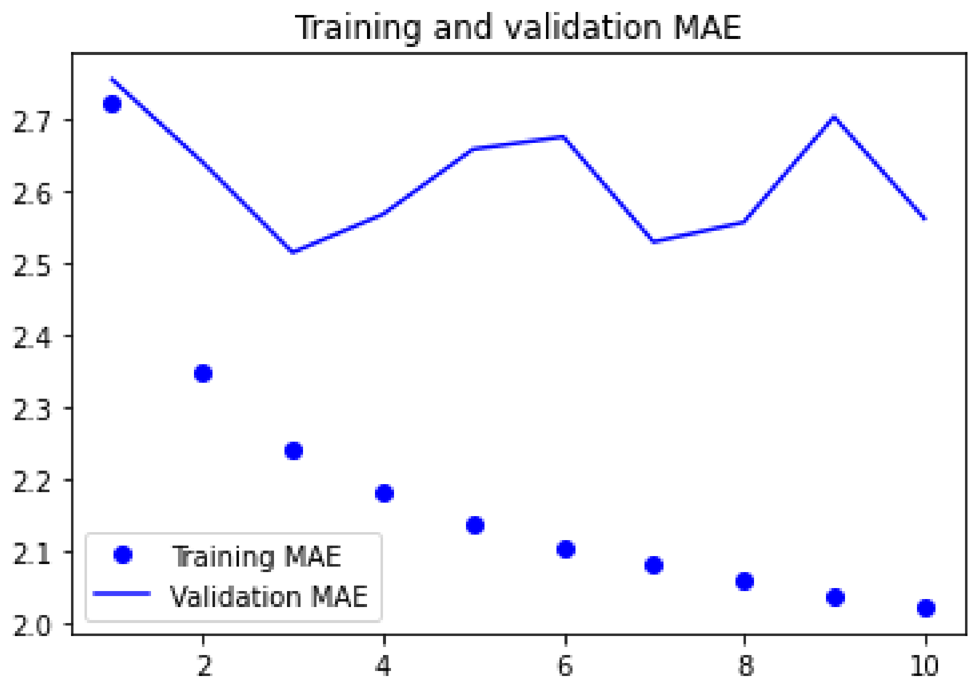
```
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    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 [==============================] - 11s 12ms/step - loss:
12.4105 - mae: 2.7234 - val_loss: 11.8392 - val_mae: 2.7181
Epoch 2/10
819/819 [==============================] - 10s 12ms/step - loss:
8.8708 - mae: 2.3413 - val_loss: 10.1860 - val_mae: 2.5069
Epoch 3/10
819/819 [==============================] - 10s 12ms/step - loss:
8.1231 - mae: 2.2427 - val_loss: 14.7207 - val_mae: 3.0533
Epoch 4/10
819/819 [==============================] - 10s 12ms/step - loss:
7.7080 - mae: 2.1850 - val_loss: 10.5407 - val_mae: 2.5550
Epoch 5/10
819/819 [==============================] - 9s 11ms/step - loss: 7.3661
- mae: 2.1381 - val_loss: 11.0491 - val_mae: 2.6141
Epoch 6/10
819/819 [==============================] - 10s 12ms/step - loss:
7.1142 - mae: 2.1019 - val_loss: 13.6564 - val_mae: 2.9179
Epoch 7/10
819/819 [==============================] - 10s 12ms/step - loss:
6.9253 - mae: 2.0728 - val_loss: 10.7469 - val_mae: 2.5723
Epoch 8/10
819/819 [==============================] - 9s 10ms/step - loss: 6.7577
- mae: 2.0489 - val_loss: 10.9049 - val_mae: 2.5911
Epoch 9/10
819/819 [==============================] - 10s 13ms/step - loss:
6.6401 - mae: 2.0313 - val_loss: 11.6349 - val_mae: 2.6864
Epoch 10/10
819/819 [==============================] - 10s 13ms/step - loss:
6.5179 - mae: 2.0129 - val_loss: 10.8114 - val_mae: 2.5814
405/405 [==============================] - 3s 6ms/step - loss: 11.1628
- mae: 2.6364
Test MAE: 2.64
```

Plotting results

```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

```python
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 [==============================] - 27s 32ms/step - loss:
21.9316 - mae: 3.6816 - val_loss: 15.9259 - val_mae: 3.1425
Epoch 2/10
819/819 [==============================] - 26s 31ms/step - loss:
14.9719 - mae: 3.0605 - val_loss: 14.3273 - val_mae: 2.9993
Epoch 3/10
819/819 [==============================] - 25s 30ms/step - loss:
13.5668 - mae: 2.9080 - val_loss: 16.6880 - val_mae: 3.2202
Epoch 4/10
819/819 [==============================] - 26s 31ms/step - loss:
12.7099 - mae: 2.8100 - val_loss: 15.0712 - val_mae: 3.0562
Epoch 5/10
819/819 [==============================] - 26s 31ms/step - loss:
12.1109 - mae: 2.7422 - val_loss: 15.5567 - val_mae: 3.1106
Epoch 6/10
819/819 [==============================] - 25s 30ms/step - loss:
11.6098 - mae: 2.6841 - val_loss: 17.7714 - val_mae: 3.3496
Epoch 7/10
819/819 [==============================] - 25s 31ms/step - loss:
11.2634 - mae: 2.6417 - val_loss: 14.0368 - val_mae: 2.9612
Epoch 8/10
819/819 [==============================] - 25s 30ms/step - loss:
10.9063 - mae: 2.6007 - val_loss: 15.1358 - val_mae: 3.0681
Epoch 9/10
819/819 [==============================] - 24s 30ms/step - loss:
10.6410 - mae: 2.5698 - val_loss: 13.9195 - val_mae: 2.9408
Epoch 10/10
819/819 [==============================] - 24s 30ms/step - loss:
10.3716 - mae: 2.5394 - val_loss: 14.2042 - val_mae: 2.9799
405/405 [==============================] - 5s 12ms/step - loss:
15.7676 - mae: 3.1169 1s - loss: 15.8423 - mae:
Test MAE: 3.12
```

A first recurrent baseline

A simple LSTM-based model

```
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 [==============================] - 46s 54ms/step - loss:
43.3232 - mae: 4.8068 - val_loss: 13.1430 - val_mae: 2.7529
Epoch 2/10
819/819 [==============================] - 44s 53ms/step - loss:
11.0373 - mae: 2.5823 - val_loss: 10.1156 - val_mae: 2.4768
Epoch 3/10
819/819 [==============================] - 44s 54ms/step - loss:
9.8226 - mae: 2.4410 - val_loss: 10.5388 - val_mae: 2.4865
Epoch 4/10
819/819 [==============================] - 44s 54ms/step - loss:
9.3841 - mae: 2.3772 - val_loss: 10.1379 - val_mae: 2.4779
Epoch 5/10
819/819 [==============================] - 44s 54ms/step - loss:
9.0988 - mae: 2.3386 - val_loss: 9.8069 - val_mae: 2.4231
Epoch 6/10
819/819 [==============================] - 40s 49ms/step - loss:
8.8600 - mae: 2.3068 - val_loss: 10.4133 - val_mae: 2.4807
Epoch 7/10
819/819 [==============================] - 43s 53ms/step - loss:
8.6415 - mae: 2.2779 - val_loss: 10.6433 - val_mae: 2.4821
Epoch 8/10
819/819 [==============================] - 44s 54ms/step - loss:
8.4358 - mae: 2.2511 - val_loss: 9.9725 - val_mae: 2.4400
Epoch 9/10
819/819 [==============================] - 44s 53ms/step - loss:
8.3101 - mae: 2.2330 - val_loss: 9.9868 - val_mae: 2.4402
Epoch 10/10
819/819 [==============================] - 42s 51ms/step - loss:
8.1368 - mae: 2.2096 - val_loss: 9.9639 - val_mae: 2.4537
405/405 [==============================] - 7s 16ms/step - loss:
11.1049 - mae: 2.6216
Test MAE: 2.62
```

Understanding recurrent neural networks

NumPy implementation of a simple RNN

```
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)
```

A recurrent layer in Keras

An RNN layer that can process sequences of any length

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

An RNN layer that returns only its last output step

```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
print(outputs.shape)
```

```
(None, 16)
```

An RNN layer that returns its full output sequence

```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
print(outputs.shape)
```

```
(None, 120, 16)
```

Stacking RNN layers

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

Advanced use of recurrent neural networks

Using recurrent dropout to fight overfitting

Training and evaluating a dropout-regularized LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(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=20,
                    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 1/20
819/819 [==============================] - 77s 93ms/step - loss:
27.9374 - mae: 3.8968 - val_loss: 9.9004 - val_mae: 2.4469
Epoch 2/20
819/819 [==============================] - 76s 93ms/step - loss:
14.8168 - mae: 2.9870 - val_loss: 9.2326 - val_mae: 2.3682
Epoch 3/20
819/819 [==============================] - 76s 93ms/step - loss:
13.8818 - mae: 2.8935 - val_loss: 9.3725 - val_mae: 2.3906
Epoch 4/20
819/819 [==============================] - 76s 93ms/step - loss:
13.1789 - mae: 2.8167 - val_loss: 9.6347 - val_mae: 2.4296
Epoch 5/20
819/819 [==============================] - 76s 93ms/step - loss:
12.7742 - mae: 2.7725 - val_loss: 9.6726 - val_mae: 2.4363
Epoch 6/20
819/819 [==============================] - 76s 93ms/step - loss:
12.3482 - mae: 2.7276 - val_loss: 9.7960 - val_mae: 2.4565
Epoch 7/20
819/819 [==============================] - 76s 93ms/step - loss:
```

```
12.0456 - mae: 2.6901 - val_loss: 9.8426 - val_mae: 2.4509
Epoch 8/20
819/819 [==============================] - 75s 92ms/step - loss:
11.8390 - mae: 2.6709 - val_loss: 10.2903 - val_mae: 2.5136
Epoch 9/20
819/819 [==============================] - 76s 93ms/step - loss:
11.6429 - mae: 2.6504 - val_loss: 9.9965 - val_mae: 2.4733
Epoch 10/20
819/819 [==============================] - 76s 92ms/step - loss:
11.5543 - mae: 2.6370 - val_loss: 9.8030 - val_mae: 2.4522
Epoch 11/20
819/819 [==============================] - 75s 92ms/step - loss:
11.4480 - mae: 2.6264 - val_loss: 9.5014 - val_mae: 2.4072
Epoch 12/20
819/819 [==============================] - 75s 92ms/step - loss:
11.2479 - mae: 2.6002 - val_loss: 9.5164 - val_mae: 2.4032
Epoch 13/20
819/819 [==============================] - 75s 92ms/step - loss:
11.1321 - mae: 2.5859 - val_loss: 9.9226 - val_mae: 2.4638
Epoch 14/20
819/819 [==============================] - 75s 92ms/step - loss:
11.0548 - mae: 2.5804 - val_loss: 9.8277 - val_mae: 2.4457
Epoch 15/20
819/819 [==============================] - 75s 91ms/step - loss:
10.8918 - mae: 2.5598 - val_loss: 9.8355 - val_mae: 2.4526
Epoch 16/20
819/819 [==============================] - 74s 91ms/step - loss:
10.8411 - mae: 2.5534 - val_loss: 9.9871 - val_mae: 2.4633
Epoch 17/20
819/819 [==============================] - 75s 91ms/step - loss:
10.7874 - mae: 2.5456 - val_loss: 10.6893 - val_mae: 2.5385
Epoch 18/20
819/819 [==============================] - 75s 91ms/step - loss:
10.7182 - mae: 2.5370 - val_loss: 10.0911 - val_mae: 2.4766
Epoch 19/20
819/819 [==============================] - 75s 92ms/step - loss:
10.6016 - mae: 2.5234 - val_loss: 10.2900 - val_mae: 2.5071
Epoch 20/20
819/819 [==============================] - 74s 91ms/step - loss:
10.4893 - mae: 2.5119 - val_loss: 10.1566 - val_mae: 2.4883
405/405 [==============================] - 7s 16ms/step - loss:
10.6684 - mae: 2.5713
Test MAE: 2.57
```

```
inputs = keras.Input(shape=(sequence_length, num_features))
x = layers.LSTM(32, recurrent_dropout=0.2, unroll=True)(inputs)
```

Stacking recurrent layers

Training and evaluating a dropout-regularized, stacked GRU model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)
(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(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=50,
                    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}")
```

Using bidirectional RNNs

Training and evaluating a bidirectional LSTM

```
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 [==============================] - 50s 57ms/step - loss:
26.8589 - mae: 3.7405 - val_loss: 10.8769 - val_mae: 2.5577
Epoch 2/10
819/819 [==============================] - 46s 56ms/step - loss:
9.6085 - mae: 2.4266 - val_loss: 10.1380 - val_mae: 2.4539
Epoch 3/10
819/819 [==============================] - 46s 56ms/step - loss:
8.7557 - mae: 2.3043 - val_loss: 10.0190 - val_mae: 2.4493
Epoch 4/10
819/819 [==============================] - 46s 56ms/step - loss:
```

```
8.3103 - mae: 2.2457 - val_loss: 9.9648 - val_mae: 2.4259
Epoch 5/10
819/819 [==============================] - 46s 56ms/step - loss:
7.8855 - mae: 2.1899 - val_loss: 10.8730 - val_mae: 2.5376
Epoch 6/10
819/819 [==============================] - 45s 55ms/step - loss:
7.5454 - mae: 2.1432 - val_loss: 10.6697 - val_mae: 2.5179
Epoch 7/10
819/819 [==============================] - 46s 56ms/step - loss:
7.2399 - mae: 2.0980 - val_loss: 10.3460 - val_mae: 2.4988
Epoch 8/10
819/819 [==============================] - 46s 57ms/step - loss:
6.9880 - mae: 2.0620 - val_loss: 11.0148 - val_mae: 2.5542
Epoch 9/10
819/819 [==============================] - 46s 56ms/step - loss:
6.7708 - mae: 2.0326 - val_loss: 11.1371 - val_mae: 2.5787
Epoch 10/10
819/819 [==============================] - 46s 56ms/step - loss:
6.5673 - mae: 1.9978 - val_loss: 11.2062 - val_mae: 2.5782
```

Going even further

Summary

This is a companion notebook for the book Deep Learning with Python, Second Edition. For readability, it only contains runnable code blocks and section titles, and omits everything else in the book: text paragraphs, figures, and pseudocode.

If you want to be able to follow what's going on, I recommend reading the notebook side by side with your copy of the book.

This notebook was generated for TensorFlow 2.6.

Deep learning for timeseries

Different kinds of timeseries tasks

A temperature-forecasting example

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

```
--2024-04-06 13:55:51--
https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.217.140.200,
52.217.92.38, 52.217.104.118, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|
52.217.140.200|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13565642 (13M) [application/zip]
Saving to: ‘jena_climate_2009_2016.csv.zip.1’

100%[======================================>] 13,565,642  46.7MB/s
in 0.3s

2024-04-06 13:55:52 (46.7 MB/s) - ‘jena_climate_2009_2016.csv.zip.1’
saved [13565642/13565642]

Archive:  jena_climate_2009_2016.csv.zip
replace jena_climate_2009_2016.csv? [y]es, [n]o, [A]ll, [N]one,
[r]ename:
```

Inspecting the data of the Jena weather dataset

```
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 (mmol/mol)"', '"rho (g/m**3)"', '"wv
(m/s)"', '"max. wv (m/s)"', '"wd (deg)"']
420451
```

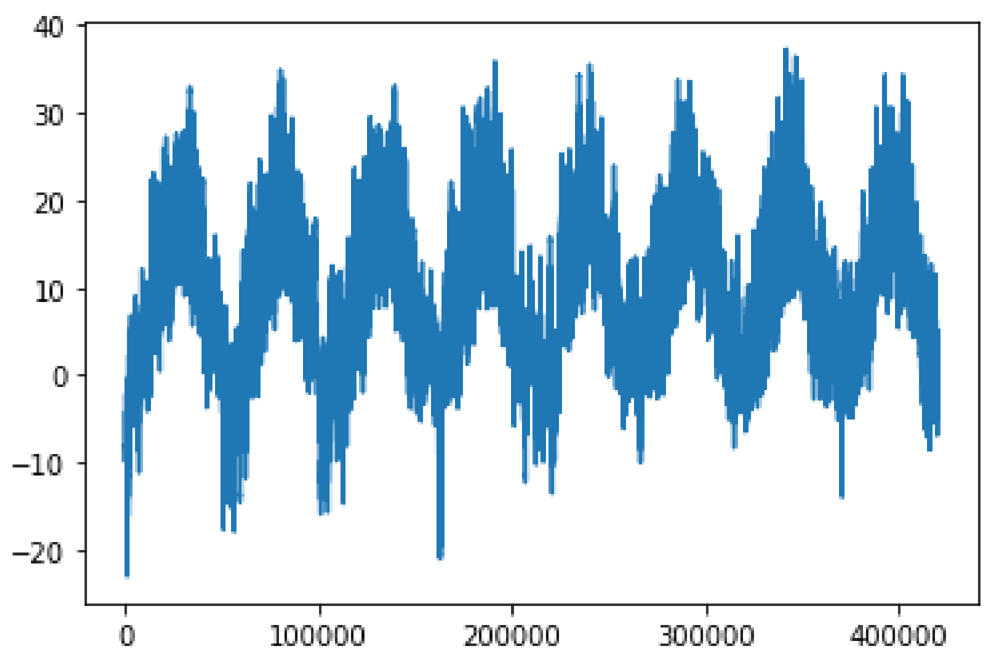
Parsing the data

```
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

```
from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)
```

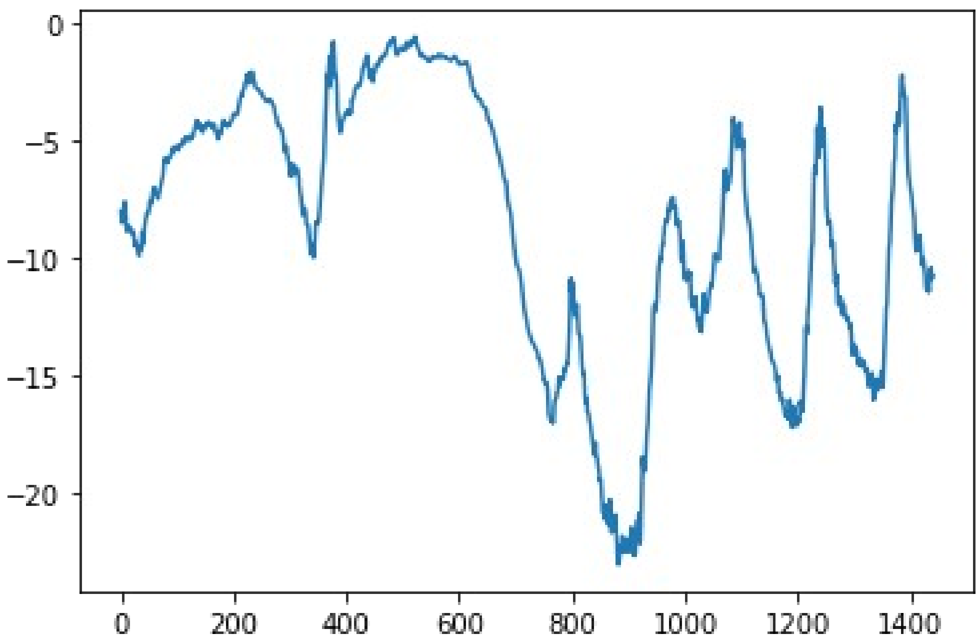
```
[<matplotlib.lines.Line2D at 0x7f9189eab518>]
```



Plotting the first 10 days of the temperature timeseries

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

```
[<matplotlib.lines.Line2D at 0x7f9181d9edd8>]
```



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

```
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

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

```python
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

```python
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

```
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

```
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
```

Let's try a basic machine-learning model

Training and evaluating a densely connected model

```
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(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
```

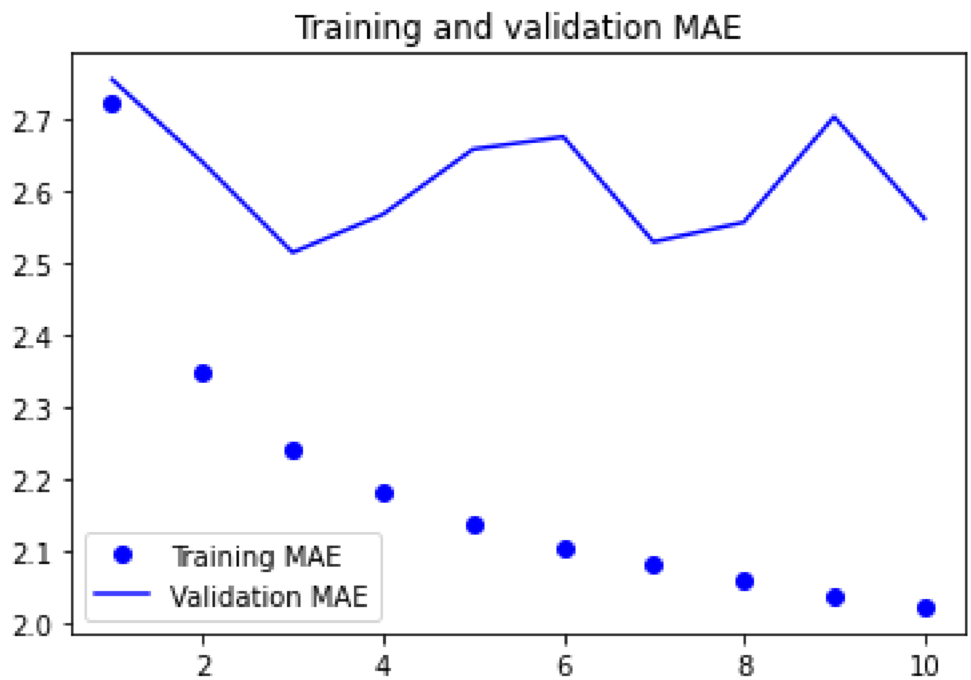
```
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    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 [==============================] - 11s 12ms/step - loss: 13.4556 - mae: 2.8390 - val_loss: 12.0676 - val_mae: 2.7676
Epoch 2/10
819/819 [==============================] - 10s 12ms/step - loss: 9.5329 - mae: 2.4264 - val_loss: 11.4273 - val_mae: 2.6859
Epoch 3/10
819/819 [==============================] - 10s 12ms/step - loss: 8.6976 - mae: 2.3191 - val_loss: 10.2813 - val_mae: 2.5337
Epoch 4/10
819/819 [==============================] - 10s 12ms/step - loss: 8.1518 - mae: 2.2470 - val_loss: 11.5264 - val_mae: 2.6956
Epoch 5/10
819/819 [==============================] - 10s 12ms/step - loss: 7.7940 - mae: 2.1960 - val_loss: 13.4233 - val_mae: 2.9176
Epoch 6/10
819/819 [==============================] - 10s 12ms/step - loss: 7.5070 - mae: 2.1577 - val_loss: 11.7076 - val_mae: 2.7153
Epoch 7/10
819/819 [==============================] - 10s 12ms/step - loss: 7.2974 - mae: 2.1275 - val_loss: 10.9852 - val_mae: 2.6277
Epoch 8/10
819/819 [==============================] - 10s 12ms/step - loss: 7.1314 - mae: 2.1043 - val_loss: 10.7224 - val_mae: 2.5963
Epoch 9/10
819/819 [==============================] - 10s 12ms/step - loss: 6.9663 - mae: 2.0805 - val_loss: 11.1391 - val_mae: 2.6518
Epoch 10/10
819/819 [==============================] - 10s 12ms/step - loss: 6.8452 - mae: 2.0607 - val_loss: 11.7002 - val_mae: 2.7126
405/405 [==============================] - 3s 8ms/step - loss: 11.0869 - mae: 2.6240
Test MAE: 2.62
```

Plotting results

```
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

```
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 [==============================] - 27s 32ms/step - loss:
21.9316 - mae: 3.6816 - val_loss: 15.9259 - val_mae: 3.1425
Epoch 2/10
819/819 [==============================] - 26s 31ms/step - loss:
14.9719 - mae: 3.0605 - val_loss: 14.3273 - val_mae: 2.9993
Epoch 3/10
819/819 [==============================] - 25s 30ms/step - loss:
13.5668 - mae: 2.9080 - val_loss: 16.6880 - val_mae: 3.2202
Epoch 4/10
819/819 [==============================] - 26s 31ms/step - loss:
12.7099 - mae: 2.8100 - val_loss: 15.0712 - val_mae: 3.0562
Epoch 5/10
819/819 [==============================] - 26s 31ms/step - loss:
12.1109 - mae: 2.7422 - val_loss: 15.5567 - val_mae: 3.1106
Epoch 6/10
819/819 [==============================] - 25s 30ms/step - loss:
11.6098 - mae: 2.6841 - val_loss: 17.7714 - val_mae: 3.3496
Epoch 7/10
819/819 [==============================] - 25s 31ms/step - loss:
11.2634 - mae: 2.6417 - val_loss: 14.0368 - val_mae: 2.9612
Epoch 8/10
819/819 [==============================] - 25s 30ms/step - loss:
10.9063 - mae: 2.6007 - val_loss: 15.1358 - val_mae: 3.0681
Epoch 9/10
819/819 [==============================] - 24s 30ms/step - loss:
10.6410 - mae: 2.5698 - val_loss: 13.9195 - val_mae: 2.9408
Epoch 10/10
819/819 [==============================] - 24s 30ms/step - loss:
10.3716 - mae: 2.5394 - val_loss: 14.2042 - val_mae: 2.9799
405/405 [==============================] - 5s 12ms/step - loss:
15.7676 - mae: 3.1169 1s - loss: 15.8423 - mae:
Test MAE: 3.12
```

A first recurrent baseline

A simple LSTM-based model

```
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 [==============================] - 46s 54ms/step - loss:
43.3232 - mae: 4.8068 - val_loss: 13.1430 - val_mae: 2.7529
Epoch 2/10
819/819 [==============================] - 44s 53ms/step - loss:
11.0373 - mae: 2.5823 - val_loss: 10.1156 - val_mae: 2.4768
Epoch 3/10
819/819 [==============================] - 44s 54ms/step - loss:
9.8226 - mae: 2.4410 - val_loss: 10.5388 - val_mae: 2.4865
Epoch 4/10
819/819 [==============================] - 44s 54ms/step - loss:
9.3841 - mae: 2.3772 - val_loss: 10.1379 - val_mae: 2.4779
Epoch 5/10
819/819 [==============================] - 44s 54ms/step - loss:
9.0988 - mae: 2.3386 - val_loss: 9.8069 - val_mae: 2.4231
Epoch 6/10
819/819 [==============================] - 40s 49ms/step - loss:
8.8600 - mae: 2.3068 - val_loss: 10.4133 - val_mae: 2.4807
Epoch 7/10
819/819 [==============================] - 43s 53ms/step - loss:
8.6415 - mae: 2.2779 - val_loss: 10.6433 - val_mae: 2.4821
Epoch 8/10
819/819 [==============================] - 44s 54ms/step - loss:
8.4358 - mae: 2.2511 - val_loss: 9.9725 - val_mae: 2.4400
Epoch 9/10
819/819 [==============================] - 44s 53ms/step - loss:
8.3101 - mae: 2.2330 - val_loss: 9.9868 - val_mae: 2.4402
Epoch 10/10
819/819 [==============================] - 42s 51ms/step - loss:
8.1368 - mae: 2.2096 - val_loss: 9.9639 - val_mae: 2.4537
405/405 [==============================] - 7s 16ms/step - loss:
11.1049 - mae: 2.6216
Test MAE: 2.62
```

Understanding recurrent neural networks

NumPy implementation of a simple RNN

```
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)
```

A recurrent layer in Keras

An RNN layer that can process sequences of any length

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

An RNN layer that returns only its last output step

```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
print(outputs.shape)
```

```
(None, 16)
```

An RNN layer that returns its full output sequence

```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
print(outputs.shape)
```

```
(None, 120, 16)
```

Stacking RNN layers

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

Advanced use of recurrent neural networks

Using recurrent dropout to fight overfitting

Training and evaluating a dropout-regularized LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(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=50,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

```
Epoch 1/50
819/819 [==============================] - 77s 92ms/step - loss:
26.6721 - mae: 3.8142 - val_loss: 9.7374 - val_mae: 2.4227
Epoch 2/50
819/819 [==============================] - 76s 93ms/step - loss:
14.6546 - mae: 2.9761 - val_loss: 9.9463 - val_mae: 2.4577
Epoch 3/50
819/819 [==============================] - 76s 93ms/step - loss:
13.8488 - mae: 2.8826 - val_loss: 9.5585 - val_mae: 2.4052
Epoch 4/50
819/819 [==============================] - 76s 92ms/step - loss:
13.1990 - mae: 2.8171 - val_loss: 9.6851 - val_mae: 2.4222
Epoch 5/50
819/819 [==============================] - 75s 92ms/step - loss:
12.6733 - mae: 2.7613 - val_loss: 9.6745 - val_mae: 2.4037
Epoch 6/50
819/819 [==============================] - 76s 93ms/step - loss:
12.2199 - mae: 2.7111 - val_loss: 9.8409 - val_mae: 2.4285
Epoch 7/50
819/819 [==============================] - 76s 93ms/step - loss:
11.8653 - mae: 2.6732 - val_loss: 9.6984 - val_mae: 2.4166
Epoch 8/50
```

```
819/819 [==============================] - 76s 93ms/step - loss:
11.6971 - mae: 2.6553 - val_loss: 9.6553 - val_mae: 2.4069
Epoch 9/50
819/819 [==============================] - 76s 93ms/step - loss:
11.4774 - mae: 2.6295 - val_loss: 9.8494 - val_mae: 2.4360
Epoch 10/50
819/819 [==============================] - 75s 92ms/step - loss:
11.2426 - mae: 2.6046 - val_loss: 9.6469 - val_mae: 2.4076
Epoch 11/50
819/819 [==============================] - 76s 93ms/step - loss:
11.0330 - mae: 2.5818 - val_loss: 9.7349 - val_mae: 2.4199
Epoch 12/50
819/819 [==============================] - 76s 92ms/step - loss:
10.9284 - mae: 2.5702 - val_loss: 9.6747 - val_mae: 2.4103
Epoch 13/50
819/819 [==============================] - 76s 92ms/step - loss:
10.8078 - mae: 2.5559 - val_loss: 9.8297 - val_mae: 2.4377
Epoch 14/50
819/819 [==============================] - 76s 92ms/step - loss:
10.6807 - mae: 2.5430 - val_loss: 9.8052 - val_mae: 2.4271
Epoch 15/50
819/819 [==============================] - 75s 92ms/step - loss:
10.6097 - mae: 2.5320 - val_loss: 9.8904 - val_mae: 2.4313
Epoch 16/50
819/819 [==============================] - 76s 93ms/step - loss:
10.4418 - mae: 2.5131 - val_loss: 9.7421 - val_mae: 2.4274
Epoch 17/50
819/819 [==============================] - 76s 93ms/step - loss:
10.4020 - mae: 2.5079 - val_loss: 9.8583 - val_mae: 2.4253
Epoch 18/50
819/819 [==============================] - 75s 92ms/step - loss:
10.3688 - mae: 2.5034 - val_loss: 10.0317 - val_mae: 2.4499
Epoch 19/50
819/819 [==============================] - 76s 93ms/step - loss:
10.2466 - mae: 2.4883 - val_loss: 9.7442 - val_mae: 2.4127
Epoch 20/50
819/819 [==============================] - 75s 92ms/step - loss:
10.1591 - mae: 2.4776 - val_loss: 10.1273 - val_mae: 2.4652
Epoch 21/50
819/819 [==============================] - 76s 93ms/step - loss:
10.0732 - mae: 2.4667 - val_loss: 10.2112 - val_mae: 2.4693
Epoch 22/50
819/819 [==============================] - 76s 93ms/step - loss:
9.0798 - mae: 2.3328 - val_loss: 11.0043 - val_mae: 2.5704
Epoch 46/50
 33/819 [>.............................] - ETA: 1:05 - loss: 9.2551 -
mae: 2.3617

inputs = keras.Input(shape=(sequence_length, num_features))
x = layers.LSTM(32, recurrent_dropout=0.2, unroll=True)(inputs)
```

Stacking recurrent layers

Training and evaluating a dropout-regularized, stacked GRU model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)
(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(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=50,
                    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 1/50
819/819 [==============================] - 131s 156ms/step - loss:
27.1613 - mae: 3.8323 - val_loss: 10.0783 - val_mae: 2.4570
Epoch 2/50
819/819 [==============================] - 128s 156ms/step - loss:
14.0082 - mae: 2.9013 - val_loss: 9.1958 - val_mae: 2.3494
Epoch 3/50
819/819 [==============================] - 127s 155ms/step - loss:
13.2716 - mae: 2.8232 - val_loss: 9.3608 - val_mae: 2.3819
Epoch 4/50
819/819 [==============================] - 126s 154ms/step - loss:
12.7107 - mae: 2.7614 - val_loss: 9.1774 - val_mae: 2.3492
Epoch 5/50
819/819 [==============================] - 127s 155ms/step - loss:
12.1682 - mae: 2.7077 - val_loss: 8.9101 - val_mae: 2.3236
Epoch 6/50
819/819 [==============================] - 126s 154ms/step - loss:
11.7723 - mae: 2.6567 - val_loss: 9.3252 - val_mae: 2.3737
Epoch 7/50
819/819 [==============================] - 127s 155ms/step - loss:
11.3377 - mae: 2.6135 - val_loss: 8.6502 - val_mae: 2.2910
Epoch 8/50
819/819 [==============================] - 126s 154ms/step - loss:
10.9499 - mae: 2.5693 - val_loss: 10.2197 - val_mae: 2.4828
Epoch 9/50
819/819 [==============================] - 127s 155ms/step - loss:
```

```
10.6536 - mae: 2.5364 - val_loss: 9.1094 - val_mae: 2.3487
Epoch 10/50
819/819 [==============================] - 127s 155ms/step - loss:
10.3493 - mae: 2.4992 - val_loss: 9.2771 - val_mae: 2.3743
Epoch 11/50
819/819 [==============================] - 127s 155ms/step - loss:
10.0721 - mae: 2.4671 - val_loss: 9.8222 - val_mae: 2.4354
Epoch 12/50
819/819 [==============================] - 126s 154ms/step - loss:
9.8192 - mae: 2.4342 - val_loss: 9.7284 - val_mae: 2.4265
Epoch 13/50
819/819 [==============================] - 125s 153ms/step - loss:
9.6028 - mae: 2.4083 - val_loss: 9.5791 - val_mae: 2.4078
Epoch 14/50
477/819 [================>.............] - ETA: 48s - loss: 9.4501 -
mae: 2.3885

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)


819/819 [==============================] - 126s 154ms/step - loss:
8.8992 - mae: 2.3175 - val_loss: 10.8102 - val_mae: 2.5552
Epoch 18/50
819/819 [==============================] - 126s 154ms/step - loss:
8.7391 - mae: 2.2969 - val_loss: 10.5609 - val_mae: 2.5189
Epoch 19/50
819/819 [==============================] - 126s 153ms/step - loss:
8.6250 - mae: 2.2832 - val_loss: 10.5270 - val_mae: 2.5081
Epoch 20/50
819/819 [==============================] - 126s 153ms/step - loss:
8.5010 - mae: 2.2659 - val_loss: 10.7600 - val_mae: 2.5325
Epoch 21/50
819/819 [==============================] - 125s 153ms/step - loss:
8.3740 - mae: 2.2493 - val_loss: 11.0026 - val_mae: 2.5687
Epoch 22/50
819/819 [==============================] - 126s 153ms/step - loss:
8.3240 - mae: 2.2412 - val_loss: 11.1037 - val_mae: 2.5867
Epoch 23/50
819/819 [==============================] - 126s 154ms/step - loss:
8.1384 - mae: 2.2171 - val_loss: 11.2153 - val_mae: 2.5900
Epoch 25/50
819/819 [==============================] - 125s 153ms/step - loss:
```

```
8.0914 - mae: 2.2115 - val_loss: 11.3392 - val_mae: 2.6167
Epoch 26/50
819/819 [==============================] - 123s 150ms/step - loss:
7.6870 - mae: 2.1554 - val_loss: 11.8607 - val_mae: 2.6664
Epoch 32/50
819/819 [==============================] - 125s 152ms/step - loss:
7.6550 - mae: 2.1496 - val_loss: 11.7946 - val_mae: 2.6726
Epoch 33/50
819/819 [==============================] - 124s 151ms/step - loss:
7.6237 - mae: 2.1436 - val_loss: 12.6886 - val_mae: 2.7654
Epoch 34/50
819/819 [==============================] - 124s 151ms/step - loss:
7.5552 - mae: 2.1328 - val_loss: 12.1324 - val_mae: 2.7027
Epoch 35/50
 46/819 [>.............................] - ETA: 1:48 - loss: 7.4868 -
mae: 2.1245

IOPub message rate exceeded.
The Jupyter server will temporarily stop sending output
to the client in order to avoid crashing it.
To change this limit, set the config variable
`--ServerApp.iopub_msg_rate_limit`.

Current values:
ServerApp.iopub_msg_rate_limit=1000.0 (msgs/sec)
ServerApp.rate_limit_window=3.0 (secs)


819/819 [==============================] - 124s 152ms/step - loss:
7.3348 - mae: 2.1023 - val_loss: 12.7650 - val_mae: 2.7724
Epoch 41/50
819/819 [==============================] - 125s 152ms/step - loss:
7.2537 - mae: 2.0909 - val_loss: 12.1053 - val_mae: 2.7100
Epoch 42/50
819/819 [==============================] - 125s 152ms/step - loss:
7.2700 - mae: 2.0934 - val_loss: 12.5526 - val_mae: 2.7521
Epoch 43/50
819/819 [==============================] - 123s 150ms/step - loss:
7.2507 - mae: 2.0902 - val_loss: 12.2526 - val_mae: 2.7217
Epoch 44/50
819/819 [==============================] - 124s 151ms/step - loss:
7.1851 - mae: 2.0830 - val_loss: 13.1357 - val_mae: 2.8061
Epoch 45/50
819/819 [==============================] - 124s 152ms/step - loss:
7.1653 - mae: 2.0796 - val_loss: 12.7088 - val_mae: 2.7719
Epoch 46/50
819/819 [==============================] - 122s 149ms/step - loss:
7.1076 - mae: 2.0699 - val_loss: 12.6462 - val_mae: 2.7604
Epoch 47/50
819/819 [==============================] - 125s 152ms/step - loss:
```

```
7.0634 - mae: 2.0639 - val_loss: 12.3118 - val_mae: 2.7266
Epoch 49/50
819/819 [==============================] - 125s 152ms/step - loss:
7.1151 - mae: 2.0693 - val_loss: 12.5235 - val_mae: 2.7475
Epoch 50/50
819/819 [==============================] - 125s 153ms/step - loss:
7.0129 - mae: 2.0555 - val_loss: 12.3479 - val_mae: 2.7407
405/405 [==============================] - 12s 28ms/step - loss:
9.6580 - mae: 2.4367
Test MAE: 2.44
```

Using bidirectional RNNs

Training and evaluating a bidirectional LSTM

```
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 [==============================] - 50s 57ms/step - loss:
22.9874 - mae: 3.4716 - val_loss: 10.0861 - val_mae: 2.4630
Epoch 2/10
819/819 [==============================] - 46s 56ms/step - loss:
9.4513 - mae: 2.3929 - val_loss: 9.5750 - val_mae: 2.3966
Epoch 3/10
819/819 [==============================] - 47s 57ms/step - loss:
8.4053 - mae: 2.2547 - val_loss: 9.5414 - val_mae: 2.3972
Epoch 4/10
819/819 [==============================] - 47s 57ms/step - loss:
7.8248 - mae: 2.1740 - val_loss: 9.7423 - val_mae: 2.4283
Epoch 5/10
819/819 [==============================] - 46s 56ms/step - loss:
7.3826 - mae: 2.1147 - val_loss: 10.4290 - val_mae: 2.5091
Epoch 6/10
819/819 [==============================] - 46s 57ms/step - loss:
7.0918 - mae: 2.0730 - val_loss: 10.7939 - val_mae: 2.5502
Epoch 7/10
819/819 [==============================] - 46s 56ms/step - loss:
6.8668 - mae: 2.0389 - val_loss: 10.1817 - val_mae: 2.4827
Epoch 8/10
819/819 [==============================] - 46s 56ms/step - loss:
6.6218 - mae: 2.0022 - val_loss: 10.4406 - val_mae: 2.5172
```

```
Epoch 9/10
527/819 [==================>...........] - ETA: 14s - loss: 6.4588 -
mae: 1.9810
```

Going even further

Summary

```
```

This is a companion notebook for the book Deep Learning with Python, Second Edition. For readability, it only contains runnable code blocks and section titles, and omits everything else in the book: text paragraphs, figures, and pseudocode.

If you want to be able to follow what's going on, I recommend reading the notebook side by side with your copy of the book.

This notebook was generated for TensorFlow 2.6.

Deep learning for timeseries

Different kinds of timeseries tasks

A temperature-forecasting example

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

```
--2024-04-06 13:55:51--
https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.217.140.200,
52.217.92.38, 52.217.104.118, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|
52.217.140.200|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13565642 (13M) [application/zip]
Saving to: ‘jena_climate_2009_2016.csv.zip.1’

100%[======================================>] 13,565,642  46.7MB/s
in 0.3s

2024-04-06 13:55:52 (46.7 MB/s) - ‘jena_climate_2009_2016.csv.zip.1’
saved [13565642/13565642]

Archive:  jena_climate_2009_2016.csv.zip
replace jena_climate_2009_2016.csv? [y]es, [n]o, [A]ll, [N]one,
[r]ename:
```

Inspecting the data of the Jena weather dataset

```
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 (mmol/mol)"', '"rho (g/m**3)"', '"wv
(m/s)"', '"max. wv (m/s)"', '"wd (deg)"']
420451
```

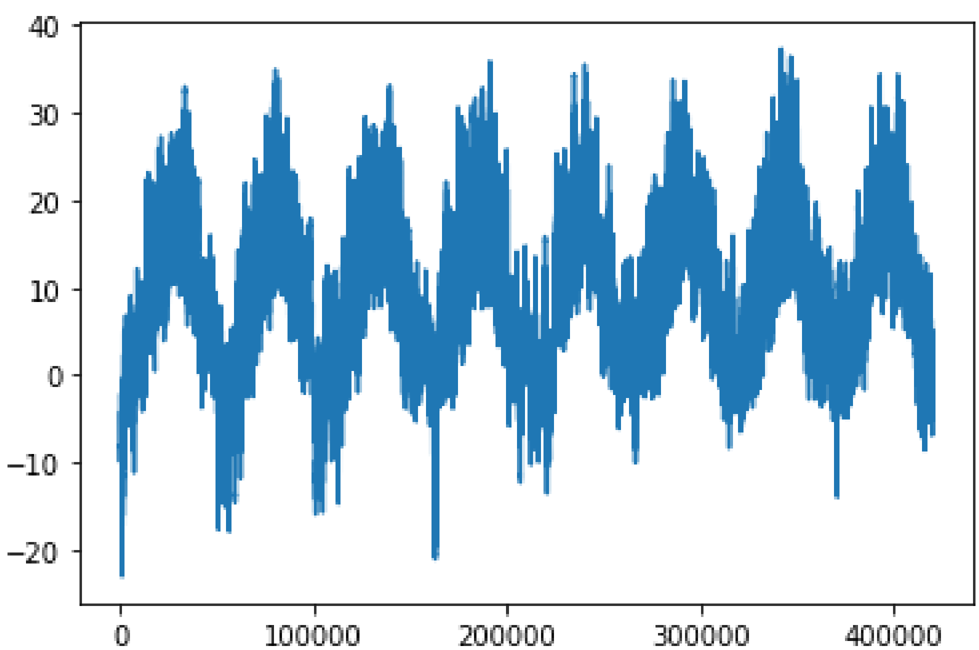
Parsing the data

```
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

```
from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)
```

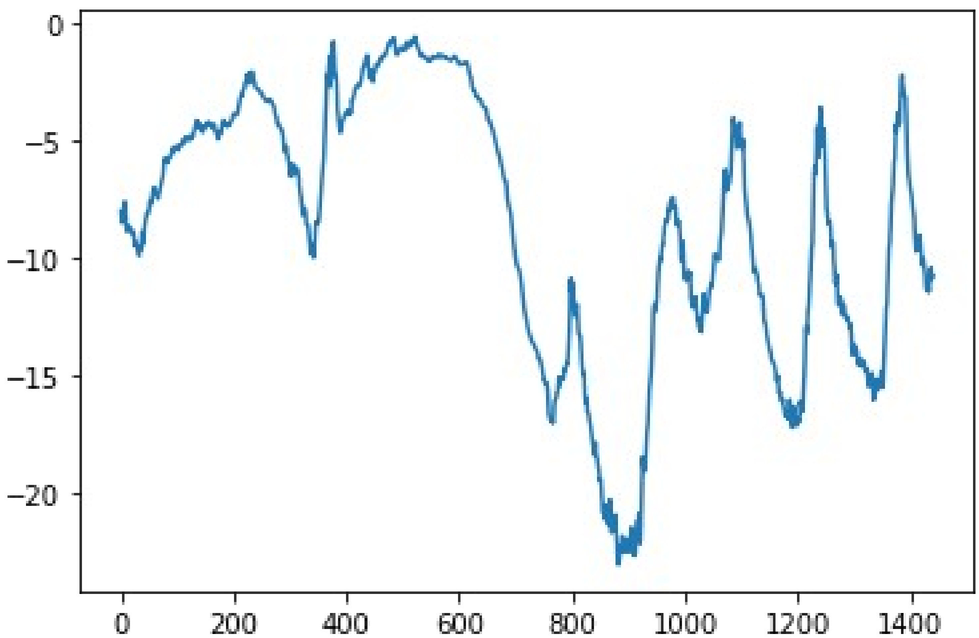
```
[<matplotlib.lines.Line2D at 0x7f3c5b658748>]
```



Plotting the first 10 days of the temperature timeseries

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

```
[<matplotlib.lines.Line2D at 0x7f3c5354ef60>]
```



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

```
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

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

```python
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

```python
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

```
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

```
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
```

Let's try a basic machine-learning model

Training and evaluating a densely connected model

```
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(16, activation="relu")(x)
outputs = layers.Dense(1)(x)
model = keras.Model(inputs, outputs)

callbacks = [
```

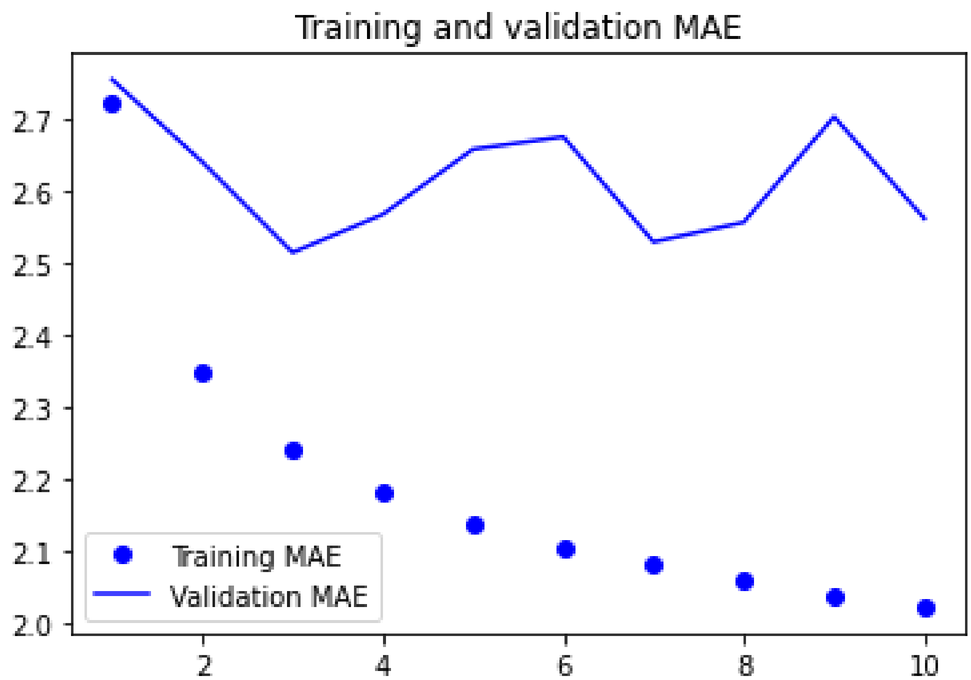
```
    keras.callbacks.ModelCheckpoint("jena_dense.keras",
                                    save_best_only=True)
]
model.compile(optimizer="rmsprop", loss="mse", metrics=["mae"])
history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    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 [==============================] - 11s 12ms/step - loss: 
12.4105 - mae: 2.7234 - val_loss: 11.8392 - val_mae: 2.7181
Epoch 2/10
819/819 [==============================] - 10s 12ms/step - loss: 
8.8708 - mae: 2.3413 - val_loss: 10.1860 - val_mae: 2.5069
Epoch 3/10
819/819 [==============================] - 10s 12ms/step - loss: 
8.1231 - mae: 2.2427 - val_loss: 14.7207 - val_mae: 3.0533
Epoch 4/10
819/819 [==============================] - 10s 12ms/step - loss: 
7.7080 - mae: 2.1850 - val_loss: 10.5407 - val_mae: 2.5550
Epoch 5/10
819/819 [==============================] - 9s 11ms/step - loss: 7.3661
- mae: 2.1381 - val_loss: 11.0491 - val_mae: 2.6141
Epoch 6/10
819/819 [==============================] - 10s 12ms/step - loss: 
7.1142 - mae: 2.1019 - val_loss: 13.6564 - val_mae: 2.9179
Epoch 7/10
819/819 [==============================] - 10s 12ms/step - loss: 
6.9253 - mae: 2.0728 - val_loss: 10.7469 - val_mae: 2.5723
Epoch 8/10
819/819 [==============================] - 9s 10ms/step - loss: 6.7577
- mae: 2.0489 - val_loss: 10.9049 - val_mae: 2.5911
Epoch 9/10
819/819 [==============================] - 10s 13ms/step - loss: 
6.6401 - mae: 2.0313 - val_loss: 11.6349 - val_mae: 2.6864
Epoch 10/10
819/819 [==============================] - 10s 13ms/step - loss: 
6.5179 - mae: 2.0129 - val_loss: 10.8114 - val_mae: 2.5814
405/405 [==============================] - 3s 6ms/step - loss: 11.1628
- mae: 2.6364
Test MAE: 2.64
```

Plotting results

```
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

```
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 [==============================] - 27s 32ms/step - loss: 
21.9316 - mae: 3.6816 - val_loss: 15.9259 - val_mae: 3.1425
Epoch 2/10
819/819 [==============================] - 26s 31ms/step - loss: 
14.9719 - mae: 3.0605 - val_loss: 14.3273 - val_mae: 2.9993
Epoch 3/10
819/819 [==============================] - 25s 30ms/step - loss: 
13.5668 - mae: 2.9080 - val_loss: 16.6880 - val_mae: 3.2202
Epoch 4/10
819/819 [==============================] - 26s 31ms/step - loss: 
12.7099 - mae: 2.8100 - val_loss: 15.0712 - val_mae: 3.0562
Epoch 5/10
819/819 [==============================] - 26s 31ms/step - loss: 
12.1109 - mae: 2.7422 - val_loss: 15.5567 - val_mae: 3.1106
Epoch 6/10
819/819 [==============================] - 25s 30ms/step - loss: 
11.6098 - mae: 2.6841 - val_loss: 17.7714 - val_mae: 3.3496
Epoch 7/10
819/819 [==============================] - 25s 31ms/step - loss: 
11.2634 - mae: 2.6417 - val_loss: 14.0368 - val_mae: 2.9612
Epoch 8/10
819/819 [==============================] - 25s 30ms/step - loss: 
10.9063 - mae: 2.6007 - val_loss: 15.1358 - val_mae: 3.0681
Epoch 9/10
819/819 [==============================] - 24s 30ms/step - loss: 
10.6410 - mae: 2.5698 - val_loss: 13.9195 - val_mae: 2.9408
Epoch 10/10
819/819 [==============================] - 24s 30ms/step - loss: 
10.3716 - mae: 2.5394 - val_loss: 14.2042 - val_mae: 2.9799
405/405 [==============================] - 5s 12ms/step - loss: 
15.7676 - mae: 3.1169 1s - loss: 15.8423 - mae: 
Test MAE: 3.12
```

A first recurrent baseline

A simple LSTM-based model

```
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 [==============================] - 46s 54ms/step - loss:
43.3232 - mae: 4.8068 - val_loss: 13.1430 - val_mae: 2.7529
Epoch 2/10
819/819 [==============================] - 44s 53ms/step - loss:
11.0373 - mae: 2.5823 - val_loss: 10.1156 - val_mae: 2.4768
Epoch 3/10
819/819 [==============================] - 44s 54ms/step - loss:
9.8226 - mae: 2.4410 - val_loss: 10.5388 - val_mae: 2.4865
Epoch 4/10
819/819 [==============================] - 44s 54ms/step - loss:
9.3841 - mae: 2.3772 - val_loss: 10.1379 - val_mae: 2.4779
Epoch 5/10
819/819 [==============================] - 44s 54ms/step - loss:
9.0988 - mae: 2.3386 - val_loss: 9.8069 - val_mae: 2.4231
Epoch 6/10
819/819 [==============================] - 40s 49ms/step - loss:
8.8600 - mae: 2.3068 - val_loss: 10.4133 - val_mae: 2.4807
Epoch 7/10
819/819 [==============================] - 43s 53ms/step - loss:
8.6415 - mae: 2.2779 - val_loss: 10.6433 - val_mae: 2.4821
Epoch 8/10
819/819 [==============================] - 44s 54ms/step - loss:
8.4358 - mae: 2.2511 - val_loss: 9.9725 - val_mae: 2.4400
Epoch 9/10
819/819 [==============================] - 44s 53ms/step - loss:
8.3101 - mae: 2.2330 - val_loss: 9.9868 - val_mae: 2.4402
Epoch 10/10
819/819 [==============================] - 42s 51ms/step - loss:
8.1368 - mae: 2.2096 - val_loss: 9.9639 - val_mae: 2.4537
405/405 [==============================] - 7s 16ms/step - loss:
11.1049 - mae: 2.6216
Test MAE: 2.62
```

Understanding recurrent neural networks

NumPy implementation of a simple RNN

```
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)
```

A recurrent layer in Keras

An RNN layer that can process sequences of any length

```
num_features = 14
inputs = keras.Input(shape=(None, num_features))
outputs = layers.SimpleRNN(16)(inputs)
```

An RNN layer that returns only its last output step

```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=False)(inputs)
print(outputs.shape)
```

```
(None, 16)
```

An RNN layer that returns its full output sequence

```
num_features = 14
steps = 120
inputs = keras.Input(shape=(steps, num_features))
outputs = layers.SimpleRNN(16, return_sequences=True)(inputs)
print(outputs.shape)
```

```
(None, 120, 16)
```

Stacking RNN layers

```
inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(16, return_sequences=True)(inputs)
x = layers.SimpleRNN(16, return_sequences=True)(x)
outputs = layers.SimpleRNN(16)(x)
```

Advanced use of recurrent neural networks

Using recurrent dropout to fight overfitting

Training and evaluating a dropout-regularized LSTM

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.LSTM(32, recurrent_dropout=0.25)(inputs)
x = layers.Dropout(0.5)(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=20,
                    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 1/20
819/819 [==============================] - 77s 93ms/step - loss:
27.9374 - mae: 3.8968 - val_loss: 9.9004 - val_mae: 2.4469
Epoch 2/20
819/819 [==============================] - 76s 93ms/step - loss:
14.8168 - mae: 2.9870 - val_loss: 9.2326 - val_mae: 2.3682
Epoch 3/20
819/819 [==============================] - 76s 93ms/step - loss:
13.8818 - mae: 2.8935 - val_loss: 9.3725 - val_mae: 2.3906
Epoch 4/20
819/819 [==============================] - 76s 93ms/step - loss:
13.1789 - mae: 2.8167 - val_loss: 9.6347 - val_mae: 2.4296
Epoch 5/20
819/819 [==============================] - 76s 93ms/step - loss:
12.7742 - mae: 2.7725 - val_loss: 9.6726 - val_mae: 2.4363
Epoch 6/20
819/819 [==============================] - 76s 93ms/step - loss:
12.3482 - mae: 2.7276 - val_loss: 9.7960 - val_mae: 2.4565
Epoch 7/20
819/819 [==============================] - 76s 93ms/step - loss:
```

```
12.0456 - mae: 2.6901 - val_loss: 9.8426 - val_mae: 2.4509
Epoch 8/20
819/819 [==============================] - 75s 92ms/step - loss:
11.8390 - mae: 2.6709 - val_loss: 10.2903 - val_mae: 2.5136
Epoch 9/20
819/819 [==============================] - 76s 93ms/step - loss:
11.6429 - mae: 2.6504 - val_loss: 9.9965 - val_mae: 2.4733
Epoch 10/20
819/819 [==============================] - 76s 92ms/step - loss:
11.5543 - mae: 2.6370 - val_loss: 9.8030 - val_mae: 2.4522
Epoch 11/20
819/819 [==============================] - 75s 92ms/step - loss:
11.4480 - mae: 2.6264 - val_loss: 9.5014 - val_mae: 2.4072
Epoch 12/20
819/819 [==============================] - 75s 92ms/step - loss:
11.2479 - mae: 2.6002 - val_loss: 9.5164 - val_mae: 2.4032
Epoch 13/20
819/819 [==============================] - 75s 92ms/step - loss:
11.1321 - mae: 2.5859 - val_loss: 9.9226 - val_mae: 2.4638
Epoch 14/20
819/819 [==============================] - 75s 92ms/step - loss:
11.0548 - mae: 2.5804 - val_loss: 9.8277 - val_mae: 2.4457
Epoch 15/20
819/819 [==============================] - 75s 91ms/step - loss:
10.8918 - mae: 2.5598 - val_loss: 9.8355 - val_mae: 2.4526
Epoch 16/20
819/819 [==============================] - 74s 91ms/step - loss:
10.8411 - mae: 2.5534 - val_loss: 9.9871 - val_mae: 2.4633
Epoch 17/20
819/819 [==============================] - 75s 91ms/step - loss:
10.7874 - mae: 2.5456 - val_loss: 10.6893 - val_mae: 2.5385
Epoch 18/20
819/819 [==============================] - 75s 91ms/step - loss:
10.7182 - mae: 2.5370 - val_loss: 10.0911 - val_mae: 2.4766
Epoch 19/20
819/819 [==============================] - 75s 92ms/step - loss:
10.6016 - mae: 2.5234 - val_loss: 10.2900 - val_mae: 2.5071
Epoch 20/20
819/819 [==============================] - 74s 91ms/step - loss:
10.4893 - mae: 2.5119 - val_loss: 10.1566 - val_mae: 2.4883
405/405 [==============================] - 7s 16ms/step - loss:
10.6684 - mae: 2.5713
Test MAE: 2.57

inputs = keras.Input(shape=(sequence_length, num_features))
x = layers.LSTM(32, recurrent_dropout=0.2, unroll=True)(inputs)
```

Stacking recurrent layers

Training and evaluating a dropout-regularized, stacked GRU model

```
inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
x = layers.GRU(32, recurrent_dropout=0.5, return_sequences=True)
(inputs)
x = layers.GRU(32, recurrent_dropout=0.5)(x)
x = layers.Dropout(0.5)(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=50,
                    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}")
```

Using bidirectional RNNs

Training and evaluating a bidirectional LSTM

```
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 [==============================] - 50s 57ms/step - loss:
22.9874 - mae: 3.4716 - val_loss: 10.0861 - val_mae: 2.4630
Epoch 2/10
819/819 [==============================] - 46s 56ms/step - loss:
9.4513 - mae: 2.3929 - val_loss: 9.5750 - val_mae: 2.3966
Epoch 3/10
819/819 [==============================] - 47s 57ms/step - loss:
8.4053 - mae: 2.2547 - val_loss: 9.5414 - val_mae: 2.3972
Epoch 4/10
819/819 [==============================] - 47s 57ms/step - loss:
```

```
7.8248 - mae: 2.1740 - val_loss: 9.7423 - val_mae: 2.4283
Epoch 5/10
819/819 [==============================] - 46s 56ms/step - loss:
7.3826 - mae: 2.1147 - val_loss: 10.4290 - val_mae: 2.5091
Epoch 6/10
819/819 [==============================] - 46s 57ms/step - loss:
7.0918 - mae: 2.0730 - val_loss: 10.7939 - val_mae: 2.5502
Epoch 7/10
819/819 [==============================] - 46s 56ms/step - loss:
6.8668 - mae: 2.0389 - val_loss: 10.1817 - val_mae: 2.4827
Epoch 8/10
819/819 [==============================] - 46s 56ms/step - loss:
6.6218 - mae: 2.0022 - val_loss: 10.4406 - val_mae: 2.5172
Epoch 9/10
527/819 [==================>...........] - ETA: 14s - loss: 6.4588 -
mae: 1.9810
```

Going even further

Summary

```

```