Weather Forecasting Using Time **Series**

Start coding or generate with AI.

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

```
--2024-10-30 04:22:52--  https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.56.224, 3.5.12.41, 3.5.13.15, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.56.224|: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  5.45MB/s    in 2.4s

2024-10-30 04:22:55 (5.45 MB/s) - 'jena_climate_2009_2016.csv.zip' saved [13565642/13565642]

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**

```
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)
420451
```

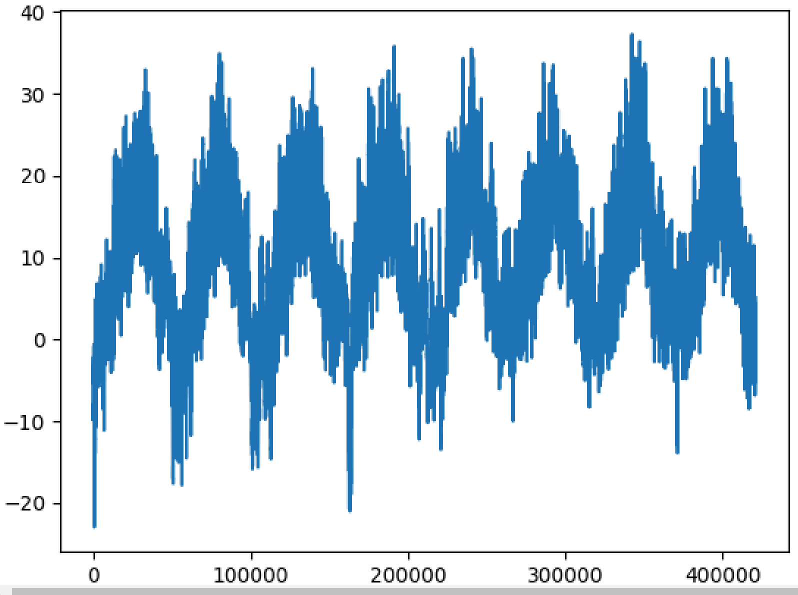
Parsing the **data**

```
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)
```

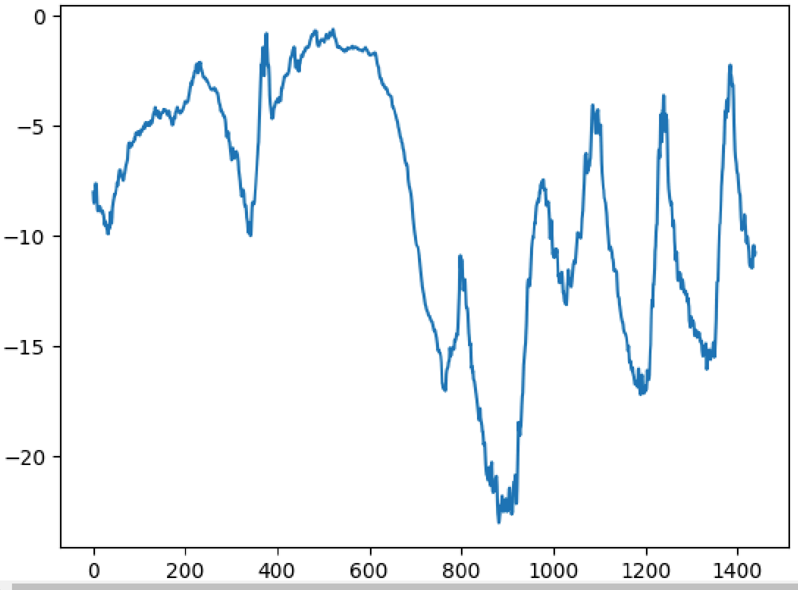
```
[<matplotlib.lines.Line2D at 0x7a0864ec57e0>]
```



Plotting the first 10 days of the temperature **timeseries**

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

```
[<matplotlib.lines.Line2D at 0x7a082da593f0>]
```



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
```

```
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**

```
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
```

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.

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(sequence_length, raw_data.shape[-1]))
# Print the shape of the inputs to debug
print("Input shape:", inputs.shape)

# Reshape the input to have a fixed batch size before flattening
x = layers.Reshape((sequence_length * raw_data.shape[-1],))(inputs)  # Reshape before Flatten
# Print the shape after reshaping to debug
print("Shape after Reshape:", x.shape)

# Now Flatten can work on this fixed shape
x = layers.Flatten()(x)
# Print the shape after flattening to debug
print("Shape after Flatten:", x.shape)

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"])

# Print the shape of the first batch of data
for batch in train_dataset.take(1):
    print("Shape of first batch:", batch[0].shape)
    break

history = model.fit(train_dataset,
                    epochs=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

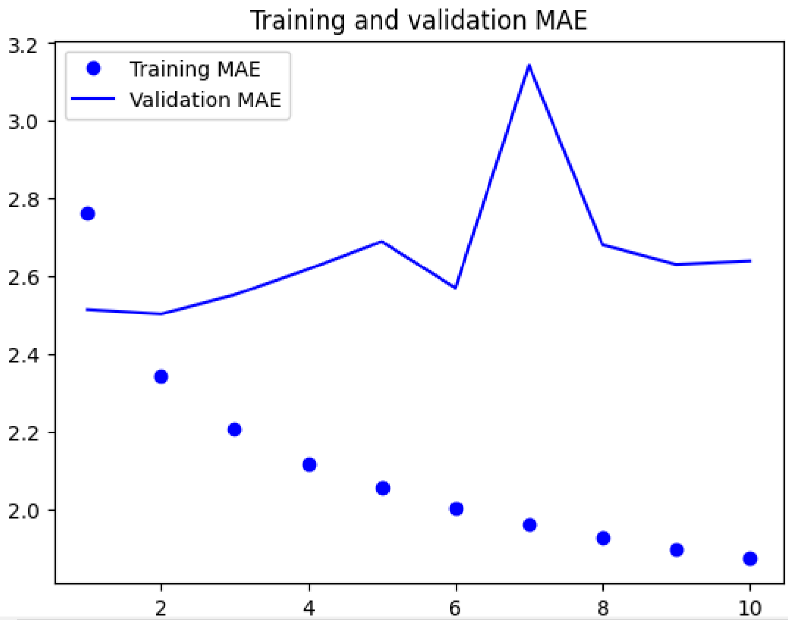
```
model = keras.models.load_model("jena_dense.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
Input shape: (None, 120, 14)
Shape after Reshape: (None, 1680)
Shape after Flatten: (None, 1680)
Shape of first batch: (256, 120, 14)
Epoch 1/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 50s 59ms/step - loss: 17.8027 - mae: 3.2093 - val_loss: 10.1617 - val_mae: 2.5151
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 48s 58ms/step - loss: 9.3192 - mae: 2.4054 - val_loss: 10.0691 - val_mae: 2.5036
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 84s 60ms/step - loss: 8.0966 - mae: 2.2420 - val_loss: 10.4647 - val_mae: 2.5529
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 81s 59ms/step - loss: 7.4000 - mae: 2.1414 - val_loss: 11.0764 - val_mae: 2.6186
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 47s 57ms/step - loss: 6.9206 - mae: 2.0757 - val_loss: 11.6703 - val_mae: 2.6896
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 49s 60ms/step - loss: 6.5586 - mae: 2.0228 - val_loss: 10.6147 - val_mae: 2.5707
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 82s 60ms/step - loss: 6.3013 - mae: 1.9782 - val_loss: 15.2822 - val_mae: 3.1429
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 54s 66ms/step - loss: 6.0895 - mae: 1.9466 - val_loss: 11.5480 - val_mae: 2.6813
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 46s 56ms/step - loss: 5.8864 - mae: 1.9118 - val_loss: 11.1099 - val_mae: 2.6310
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 90s 65ms/step - loss: 5.7145 - mae: 1.8849 - val_loss: 11.2169 - val_mae: 2.6400
405/405 ━━━━━━━━━━━━━━━━━━━━ 19s 44ms/step - loss: 11.1123 - mae: 2.6403
Test MAE: 2.64
```

Obtained a test MAE of 2.64 with densely connected model

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 ━━━━━━━━━━━━━━━━━━━━ 79s 95ms/step - loss: 31.6535 - mae: 4.2927 - val_loss: 15.4817 - val_mae: 3.0761
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 73s 88ms/step - loss: 16.3751 - mae: 3.2195 - val_loss: 15.2530 - val_mae: 3.0450
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 85s 92ms/step - loss: 14.8513 - mae: 3.0696 - val_loss: 15.9271 - val_mae: 3.1363
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 84s 94ms/step - loss: 13.7325 - mae: 2.9484 - val_loss: 16.0997 - val_mae: 3.1342
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 75s 91ms/step - loss: 12.7982 - mae: 2.8435 - val_loss: 17.2477 - val_mae: 3.2570
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 75s 91ms/step - loss: 12.1438 - mae: 2.7648 - val_loss: 15.8580 - val_mae: 3.1340
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 77s 94ms/step - loss: 11.6410 - mae: 2.7037 - val_loss: 17.5108 - val_mae: 3.2833
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 82s 94ms/step - loss: 11.2000 - mae: 2.6508 - val_loss: 17.6212 - val_mae: 3.2810
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 80s 97ms/step - loss: 10.9033 - mae: 2.6180 - val_loss: 17.0003 - val_mae: 3.2357
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 80s 94ms/step - loss: 10.6122 - mae: 2.5831 - val_loss: 15.4505 - val_mae: 3.0767
405/405 ━━━━━━━━━━━━━━━━━━━━ 20s 49ms/step - loss: 15.7218 - mae: 3.1439
Test MAE: 3.16
```

A regular 1D convultional network yielded a test MAE of 3.16 which is more than the dense layer network means it is underperforming.

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 ━━━━━━━━━━━━━━━━━━━━ 116s 140ms/step - loss: 74.6444 - mae: 6.6216 - val_loss: 13.0196 - val_mae: 2.7175
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 124s 119ms/step - loss: 12.0726 - mae: 2.6657 - val_loss: 9.1823 - val_mae: 2.3603
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 144s 121ms/step - loss: 9.6018 - mae: 2.4046 - val_loss: 9.3785 - val_mae: 2.3873
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 98s 120ms/step - loss: 8.9807 - mae: 2.3218 - val_loss: 9.4373 - val_mae: 2.3938
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 98s 120ms/step - loss: 8.5847 - mae: 2.2729 - val_loss: 10.1519 - val_mae: 2.4950
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 117s 142ms/step - loss: 8.3259 - mae: 2.2429 - val_loss: 9.8653 - val_mae: 2.4538
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 101s 123ms/step - loss: 8.1087 - mae: 2.2140 - val_loss: 9.7572 - val_mae: 2.4376
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 98s 119ms/step - loss: 7.9137 - mae: 2.1906 - val_loss: 10.3961 - val_mae: 2.4941
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 142s 120ms/step - loss: 7.7449 - mae: 2.1676 - val_loss: 10.1530 - val_mae: 2.4782
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 98s 120ms/step - loss: 7.5761 - mae: 2.1442 - val_loss: 10.4549 - val_mae: 2.5088
405/405 ━━━━━━━━━━━━━━━━━━━━ 25s 59ms/step - loss: 10.7580 - mae: 2.5776
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**

```
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.bold text

```
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 ━━━━━━━━━━━━━━━━━━━━ 100s 120ms/step - loss: 9.6215 - mae: 2.4078 - val_loss: 9.3197 - val_mae: 2.3761
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 143s 122ms/step - loss: 9.0081 - mae: 2.3247 - val_loss: 9.6388 - val_mae: 2.4186
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 141s 121ms/step - loss: 8.5587 - mae: 2.2710 - val_loss: 9.6760 - val_mae: 2.4274
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 141s 119ms/step - loss: 8.2808 - mae: 2.2359 - val_loss: 9.8074 - val_mae: 2.4536
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 141s 118ms/step - loss: 8.0602 - mae: 2.2100 - val_loss: 9.5541 - val_mae: 2.4245
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 143s 120ms/step - loss: 7.8726 - mae: 2.1845 - val_loss: 9.7469 - val_mae: 2.4400
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 99s 121ms/step - loss: 7.7013 - mae: 2.1589 - val_loss: 9.9259 - val_mae: 2.4623
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 118s 143ms/step - loss: 7.5602 - mae: 2.1368 - val_loss: 9.8476 - val_mae: 2.4552
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 122s 119ms/step - loss: 7.3967 - mae: 2.1138 - val_loss: 10.0773 - val_mae: 2.4854
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 99s 121ms/step - loss: 7.2945 - mae: 2.0969 - val_loss: 10.2410 - val_mae: 2.5064
```

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

```
405/405 ━━━━━━━━━━━━━━━━━━━━ 25s 60ms/step - loss: 10.6394 - mae: 2.5739
Test MAE: 2.58
```

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()**bold text**

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.

```
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 ━━━━━━━━━━━━━━━━━━━━ 430s 521ms/step - loss: 40.1362 - mae: 4.6250 - val_loss: 9.4018 - val_mae: 2.3815
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 426s 520ms/step - loss: 12.2479 - mae: 2.7180 - val_loss: 9.1845 - val_mae: 2.3549
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 442s 520ms/step - loss: 10.8460 - mae: 2.5586 - val_loss: 9.6278 - val_mae: 2.4252
```

```
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 443s 521ms/step - loss: 9.5548 - mae: 2.4074 - val_loss: 9.8816 - val_mae: 2.4392
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 449s 548ms/step - loss: 8.3885 - mae: 2.2478 - val_loss: 10.3977 - val_mae: 2.5253
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 450s 549ms/step - loss: 7.3162 - mae: 2.0913 - val_loss: 10.6730 - val_mae: 2.5534
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 483s 526ms/step - loss: 6.3569 - mae: 1.9409 - val_loss: 11.5902 - val_mae: 2.6305
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 427s 521ms/step - loss: 5.6971 - mae: 1.8284 - val_loss: 12.2397 - val_mae: 2.7136
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 463s 547ms/step - loss: 5.2020 - mae: 1.7385 - val_loss: 11.8498 - val_mae: 2.6687
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 425s 519ms/step - loss: 4.7635 - mae: 1.6628 - val_loss: 12.6309 - val_mae: 2.7655
405/405 ━━━━━━━━━━━━━━━━━━━━ 67s 163ms/step - loss: 10.0367 - mae: 2.4892
Test MAE: 2.49
```

Using GRU stacked RNN the test MAE reduced to even more to 2.49. 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.

```
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 ━━━━━━━━━━━━━━━━━━━━ 409s 490ms/step - loss: 41.6270 - mae: 4.7003 - val_loss: 9.8750 - val_mae: 2.4404
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 400s 488ms/step - loss: 10.9396 - mae: 2.5596 - val_loss: 10.4216 - val_mae: 2.5267
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 441s 487ms/step - loss: 8.5155 - mae: 2.2383 - val_loss: 11.6895 - val_mae: 2.6906
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 399s 487ms/step - loss: 7.1425 - mae: 2.0239 - val_loss: 13.3249 - val_mae: 2.8706
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 396s 484ms/step - loss: 6.1571 - mae: 1.8675 - val_loss: 12.1540 - val_mae: 2.7352
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 445s 488ms/step - loss: 5.5741 - mae: 1.7717 - val_loss: 12.9339 - val_mae: 2.8171
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 398s 486ms/step - loss: 5.0884 - mae: 1.6896 - val_loss: 12.5706 - val_mae: 2.7824
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 397s 484ms/step - loss: 4.6980 - mae: 1.6162 - val_loss: 12.9817 - val_mae: 2.8292
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 418s 510ms/step - loss: 4.4049 - mae: 1.5635 - val_loss: 12.8435 - val_mae: 2.8078
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 399s 486ms/step - loss: 4.1781 - mae: 1.5220 - val_loss: 13.1957 - val_mae: 2.8506
405/405 ━━━━━━━━━━━━━━━━━━━━ 76s 186ms/step - loss: 11.2363 - mae: 2.6177
Test MAE: 2.62
```

With LSTM, the test MAE is 2.62 which is little similar to GRU. Both LSTM and GRU performed similarly with slight changes.

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 ━━━━━━━━━━━━━━━━━━━━ 146s 172ms/step - loss: 48.3140 - mae: 5.1403 - val_loss: 10.6959 - val_mae: 2.5471
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 141s 172ms/step - loss: 9.7795 - mae: 2.4455 - val_loss: 10.0355 - val_mae: 2.4647
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 138s 169ms/step - loss: 8.5803 - mae: 2.2812 - val_loss: 10.5178 - val_mae: 2.5187
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 142s 169ms/step - loss: 8.0503 - mae: 2.2108 - val_loss: 10.2027 - val_mae: 2.4823
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 142s 173ms/step - loss: 7.6246 - mae: 2.1493 - val_loss: 10.6635 - val_mae: 2.5486
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 142s 174ms/step - loss: 7.2603 - mae: 2.0958 - val_loss: 11.1916 - val_mae: 2.6084
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 139s 170ms/step - loss: 6.9262 - mae: 2.0463 - val_loss: 11.0238 - val_mae: 2.5865
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 139s 166ms/step - loss: 6.7426 - mae: 2.0166 - val_loss: 11.3403 - val_mae: 2.6277
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 145s 169ms/step - loss: 6.5060 - mae: 1.9785 - val_loss: 12.3408 - val_mae: 2.7389
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 139s 169ms/step - loss: 6.3750 - mae: 1.9568 - val_loss: 11.4033 - val_mae: 2.6436
```

3. Using a combination of 1d_convnets and RNN.

A conv 1D stacked with RNN LSTM

```
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 ━━━━━━━━━━━━━━━━━━━━ 110s 126ms/step - loss: 48.5167 - mae: 5.2125 - val_loss: 15.5999 - val_mae: 3.0813
Epoch 2/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 142s 127ms/step - loss: 18.3873 - mae: 3.3253 - val_loss: 13.4201 - val_mae: 2.8709
Epoch 3/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 95s 115ms/step - loss: 16.7394 - mae: 3.1574 - val_loss: 14.5355 - val_mae: 3.0273
Epoch 4/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 102s 124ms/step - loss: 15.7749 - mae: 3.0607 - val_loss: 12.3459 - val_mae: 2.7814
```

```
Epoch 5/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 131s 110ms/step - loss: 14.7103 - mae: 2.9569 - val_loss: 12.8466 - val_mae: 2.8405
Epoch 6/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 142s 110ms/step - loss: 14.1416 - mae: 2.8902 - val_loss: 12.3610 - val_mae: 2.7874
Epoch 7/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 91s 111ms/step - loss: 13.4760 - mae: 2.8096 - val_loss: 12.3146 - val_mae: 2.7781
Epoch 8/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 94s 115ms/step - loss: 13.0084 - mae: 2.7723 - val_loss: 13.5223 - val_mae: 2.9209
Epoch 9/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 144s 117ms/step - loss: 12.5655 - mae: 2.7222 - val_loss: 13.3956 - val_mae: 2.9076
Epoch 10/10
819/819 ━━━━━━━━━━━━━━━━━━━━ 90s 110ms/step - loss: 12.2041 - mae: 2.6765 - val_loss: 14.3863 - val_mae: 2.9986
```

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

```
405/405 ━━━━━━━━━━━━━━━━━━━━ 22s 53ms/step - loss: 13.5998 - mae: 2.9043
Test MAE: 2.91
```

With combination of conv1d and RNN lstm, the model got worsened with test MAE 2.91.

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.