Deep learning for timeseries

Weather Forecasting Using Time Series

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

```
--2024-04-08 23:51:14--
https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.32.32,
54.231.169.248, 16.182.32.160, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.32.32|: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  36.4MB/s    in
0.4s

2024-04-08 23:51:15 (36.4 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)"', '"VPdef
(mbar)"', '"sh (g/kg)"', '"H2OC (mmol/mol)"', '"rho (g/m**3)"', '"wv
```

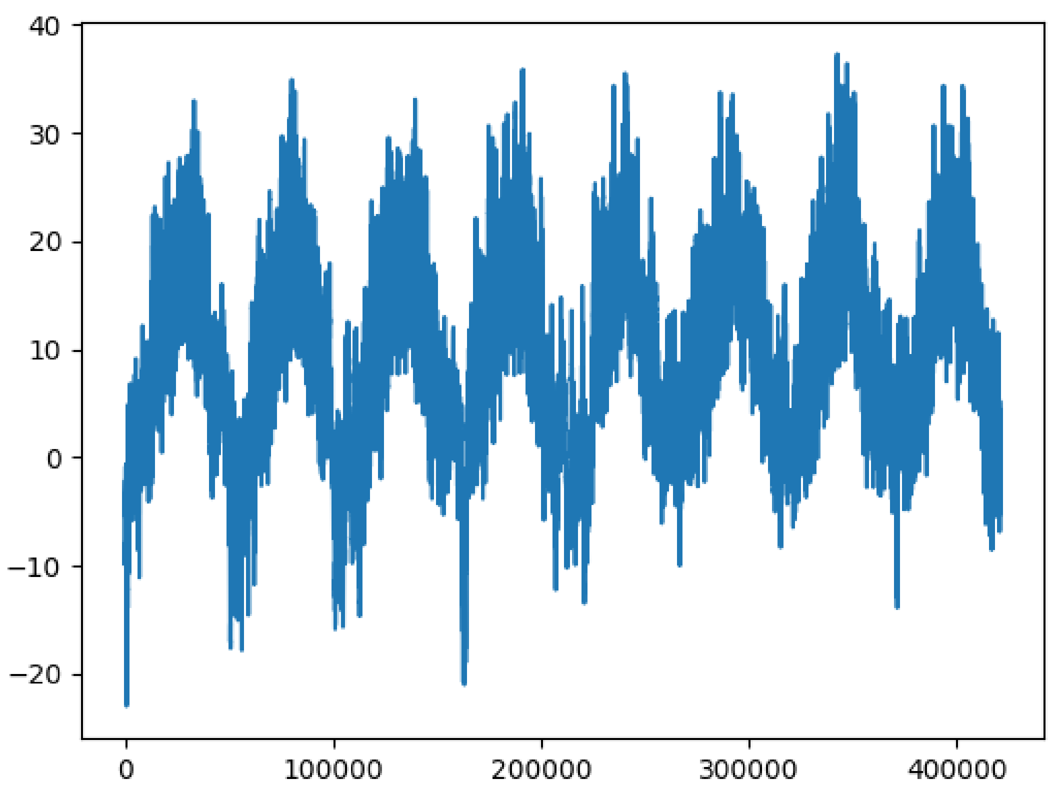
```
(m/s)"', '"max. wv (m/s)"', '"wd (deg)"']
420451
```

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[:]
```

The temperature time series is plotted

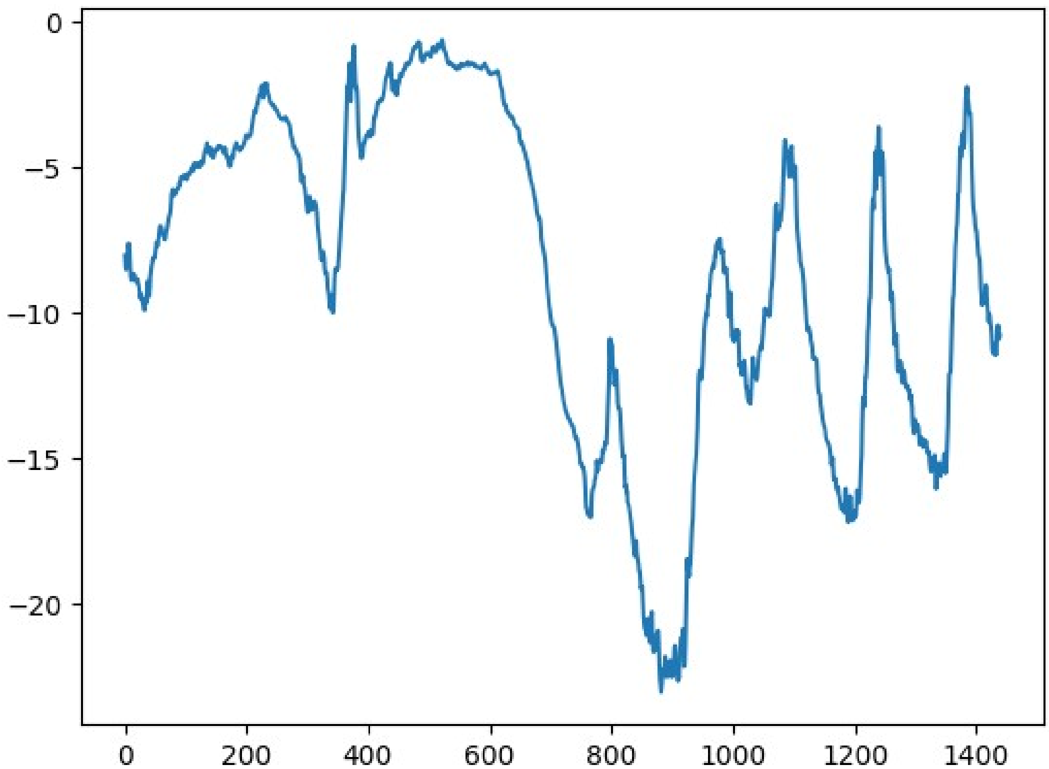
```
from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)

[<matplotlib.lines.Line2D at 0x78b735072c80>]
```



Plotting the first 10 days of the temperature timeseries

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

```
[<matplotlib.lines.Line2D at 0x78b730da7790>]
```



calculating the quantity of samples each data split will require

```
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
```

Making training, validation, and testing datasets instantiated

```
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

**Constructing and assessing a densely linked model

**featuring two dense layers and 32 units with a relu activation mechanism in the input layer.

**The model is trained using the RMSprop optimizer, which provides adaptable learning rates.

**The loss function, or mean squared error (MSE), quantifies the variation between values that were expected and those that were observed.

**During training, one statistic to keep an eye on is Mean Absolute Error (MAE), which gives information about how well the model performs on the validation set.

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

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

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

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

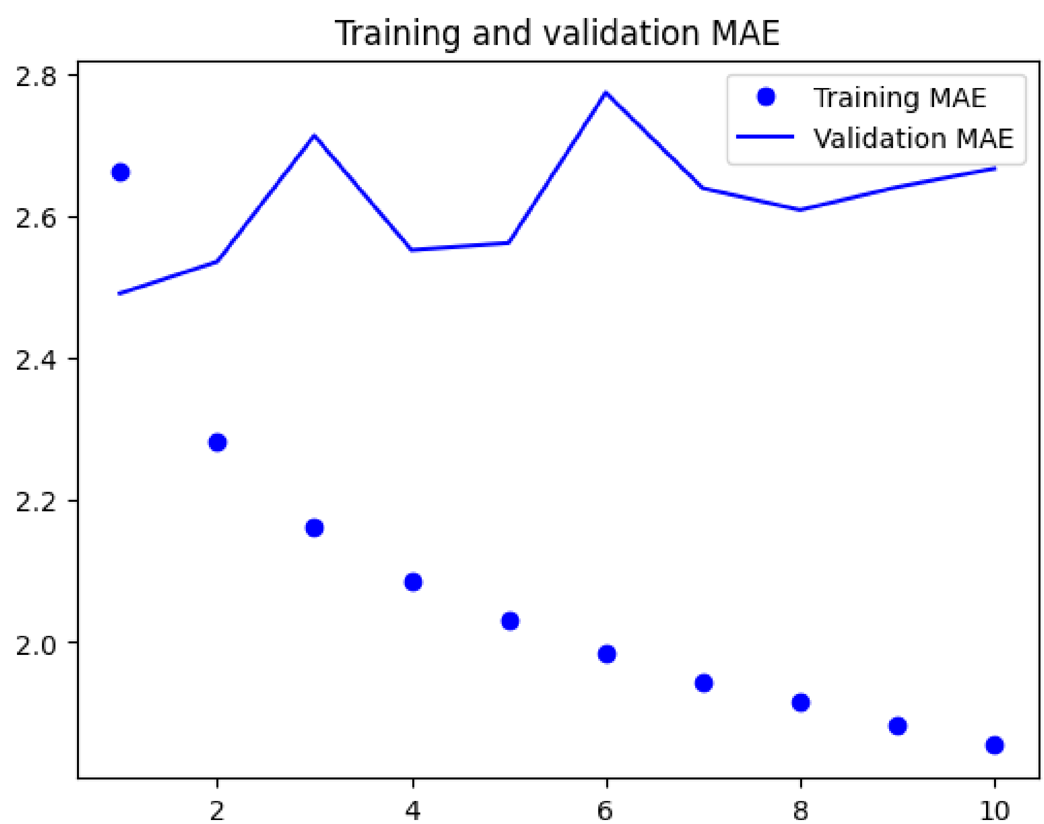
```
Epoch 1/10
819/819 [==============================] - 47s 55ms/step - loss:
11.8917 - mae: 2.6711 - val_loss: 10.1943 - val_mae: 2.5177
Epoch 2/10
819/819 [==============================] - 47s 57ms/step - loss:
8.6761 - mae: 2.3144 - val_loss: 12.1412 - val_mae: 2.7554
Epoch 3/10
819/819 [==============================] - 44s 53ms/step - loss:
7.7201 - mae: 2.1842 - val_loss: 11.3653 - val_mae: 2.6515
Epoch 4/10
819/819 [==============================] - 52s 63ms/step - loss:
7.0926 - mae: 2.0950 - val_loss: 10.6161 - val_mae: 2.5695
Epoch 5/10
819/819 [==============================] - 37s 44ms/step - loss:
6.6466 - mae: 2.0308 - val_loss: 10.9727 - val_mae: 2.6191
Epoch 6/10
819/819 [==============================] - 35s 43ms/step - loss:
6.2707 - mae: 1.9719 - val_loss: 10.8619 - val_mae: 2.6044
Epoch 7/10
819/819 [==============================] - 36s 43ms/step - loss:
5.9748 - mae: 1.9251 - val_loss: 10.7958 - val_mae: 2.5886
Epoch 8/10
819/819 [==============================] - 39s 47ms/step - loss:
5.7295 - mae: 1.8869 - val_loss: 14.5925 - val_mae: 3.0396
Epoch 9/10
```

```
819/819 [==============================] - 35s 43ms/step - loss:
5.5125 - mae: 1.8484 - val_loss: 11.3969 - val_mae: 2.6755
Epoch 10/10
819/819 [==============================] - 49s 60ms/step - loss:
5.3224 - mae: 1.8174 - val_loss: 12.4110 - val_mae: 2.7826
405/405 [==============================] - 12s 28ms/step - loss:
11.2444 - mae: 2.6289
Test MAE: 2.63
```

Obtained a test MAE of **2.62** 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 [==============================] - 15s 15ms/step - loss:
25.5756 - mae: 3.9074 - val_loss: 16.2414 - val_mae: 3.1606
Epoch 2/10
819/819 [==============================] - 13s 15ms/step - loss:
15.7580 - mae: 3.1514 - val_loss: 15.7541 - val_mae: 3.0715
Epoch 3/10
819/819 [==============================] - 12s 15ms/step - loss:
14.3575 - mae: 3.0072 - val_loss: 14.6128 - val_mae: 2.9698
Epoch 4/10
819/819 [==============================] - 12s 15ms/step - loss:
13.4928 - mae: 2.9117 - val_loss: 15.3138 - val_mae: 3.0876
Epoch 5/10
819/819 [==============================] - 12s 14ms/step - loss:
12.7442 - mae: 2.8232 - val_loss: 15.9253 - val_mae: 3.1454
Epoch 6/10
819/819 [==============================] - 12s 15ms/step - loss:
12.1946 - mae: 2.7589 - val_loss: 14.0912 - val_mae: 2.9465
Epoch 7/10
819/819 [==============================] - 12s 14ms/step - loss:
11.6940 - mae: 2.7018 - val_loss: 14.2415 - val_mae: 2.9575
Epoch 8/10
819/819 [==============================] - 12s 15ms/step - loss:
11.2607 - mae: 2.6522 - val_loss: 14.6832 - val_mae: 3.0235
Epoch 9/10
```

```
819/819 [==============================] - 13s 16ms/step - loss: 
10.9191 - mae: 2.6121 - val_loss: 15.1459 - val_mae: 3.0535
Epoch 10/10
819/819 [==============================] - 12s 14ms/step - loss: 
10.6071 - mae: 2.5753 - val_loss: 15.1200 - val_mae: 3.0549
405/405 [==============================] - 4s 9ms/step - loss: 16.5499 
- mae: 3.2040
Test MAE: 3.20
```

When compared to the dense layer network, a conventional 1D convultional network performed worse, yielding a test MAE of 3.2.

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 [==============================] - 41s 46ms/step - loss: 
39.3152 - mae: 4.5516 - val_loss: 12.1028 - val_mae: 2.6579
Epoch 2/10
819/819 [==============================] - 39s 48ms/step - loss: 
10.7773 - mae: 2.5581 - val_loss: 9.5849 - val_mae: 2.4028
Epoch 3/10
819/819 [==============================] - 38s 46ms/step - loss: 
9.7498 - mae: 2.4381 - val_loss: 9.5220 - val_mae: 2.4007
Epoch 4/10
819/819 [==============================] - 48s 58ms/step - loss: 
9.3933 - mae: 2.3930 - val_loss: 9.6857 - val_mae: 2.4200
Epoch 5/10
819/819 [==============================] - 38s 47ms/step - loss: 
9.0894 - mae: 2.3580 - val_loss: 10.0570 - val_mae: 2.4637
```

```
Epoch 6/10
819/819 [==============================] - 43s 53ms/step - loss:
8.7743 - mae: 2.3188 - val_loss: 9.8885 - val_mae: 2.4488
Epoch 7/10
819/819 [==============================] - 37s 45ms/step - loss:
8.5373 - mae: 2.2868 - val_loss: 9.7223 - val_mae: 2.4376
Epoch 8/10
819/819 [==============================] - 38s 47ms/step - loss:
8.2973 - mae: 2.2559 - val_loss: 9.9589 - val_mae: 2.4616
Epoch 9/10
819/819 [==============================] - 38s 46ms/step - loss:
8.1481 - mae: 2.2353 - val_loss: 10.1213 - val_mae: 2.4753
Epoch 10/10
819/819 [==============================] - 39s 47ms/step - loss:
7.9775 - mae: 2.2109 - val_loss: 10.5128 - val_mae: 2.5277
405/405 [==============================] - 14s 32ms/step - loss:
10.8918 - mae: 2.5568
Test MAE: 2.56
```

An essential starting point Using LSTM, an RNN was constructed, and the test MAE increased to 2.59.

Understanding recurrent neural networks

NumPy implementation of a simple RNN

```python
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

- Sequential data is processed via stacked SimpleRNN layers with increasing units (32, 32).
- Mean Absolute Error (MAE) metric and Mean Squared Error (MSE) loss are utilized with the RMSprop optimizer.

```
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 [==============================] - 39s 46ms/step - loss:
9.3965 - mae: 2.3945 - val_loss: 9.5501 - val_mae: 2.4002
Epoch 2/10
819/819 [==============================] - 38s 46ms/step - loss:
9.0597 - mae: 2.3542 - val_loss: 10.1307 - val_mae: 2.4632
Epoch 3/10
819/819 [==============================] - 47s 58ms/step - loss:
8.7603 - mae: 2.3172 - val_loss: 9.7207 - val_mae: 2.4221
Epoch 4/10
819/819 [==============================] - 38s 46ms/step - loss:
8.5038 - mae: 2.2813 - val_loss: 9.8803 - val_mae: 2.4464
Epoch 5/10
819/819 [==============================] - 40s 48ms/step - loss:
8.2237 - mae: 2.2466 - val_loss: 10.2176 - val_mae: 2.4869
Epoch 6/10
819/819 [==============================] - 39s 48ms/step - loss:
8.0488 - mae: 2.2195 - val_loss: 10.3427 - val_mae: 2.4996
Epoch 7/10
819/819 [==============================] - 39s 47ms/step - loss:
```

```
7.8953 - mae: 2.1987 - val_loss: 10.3325 - val_mae: 2.4990
Epoch 8/10
819/819 [==============================] - 40s 48ms/step - loss:
7.7707 - mae: 2.1815 - val_loss: 10.2664 - val_mae: 2.4893
Epoch 9/10
819/819 [==============================] - 48s 59ms/step - loss:
7.6633 - mae: 2.1638 - val_loss: 10.0970 - val_mae: 2.4743
Epoch 10/10
819/819 [==============================] - 40s 49ms/step - loss:
7.5882 - mae: 2.1531 - val_loss: 10.2308 - val_mae: 2.4900

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

405/405 [==============================] - 13s 30ms/step - loss:
10.9685 - mae: 2.5635
Test MAE: 2.56
```

- The MAE of a two-layer simpleRNN is 9.9.
- The error is significantly higher than that of a simple LSM model.

2. Using layer_lstm() instead of layer_gru()

Stacking RNNs with GRU and LSTM

*Stacking, dropout-regularized GRU model training and evaluation**

- The system uses two stacked GRU layers, the first of which has 64 units and the second of which has 32 units. *To avoid overfitting, a dropout layer with a dropout rate of 0.4 comes after the second GRU layer.*
  A dropout layer with a dropout rate of 0.4 comes after the second GRU layer to avoid overfitting. The Mean Squared Error (MSE) loss function, Mean Absolute Error (MAE) measure, and RMSprop optimizer are used in the compilation of the model.

```
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 [==============================] - 47s 51ms/step - loss:
21.9430 - mae: 3.4374 - val_loss: 9.1768 - val_mae: 2.3611
Epoch 2/10
819/819 [==============================] - 41s 50ms/step - loss:
11.6173 - mae: 2.6531 - val_loss: 8.8360 - val_mae: 2.3032
Epoch 3/10
819/819 [==============================] - 43s 52ms/step - loss:
10.3138 - mae: 2.4980 - val_loss: 9.3477 - val_mae: 2.3741
Epoch 4/10
819/819 [==============================] - 42s 51ms/step - loss:
9.0630 - mae: 2.3404 - val_loss: 10.0167 - val_mae: 2.4543
Epoch 5/10
819/819 [==============================] - 43s 52ms/step - loss:
7.9199 - mae: 2.1861 - val_loss: 11.1780 - val_mae: 2.6066
Epoch 6/10
819/819 [==============================] - 42s 51ms/step - loss:
6.9326 - mae: 2.0380 - val_loss: 11.6818 - val_mae: 2.6465
Epoch 7/10
819/819 [==============================] - 45s 55ms/step - loss:
6.2087 - mae: 1.9172 - val_loss: 11.6125 - val_mae: 2.6394
Epoch 8/10
819/819 [==============================] - 45s 55ms/step - loss:
5.6122 - mae: 1.8148 - val_loss: 12.4785 - val_mae: 2.7522
Epoch 9/10
819/819 [==============================] - 43s 53ms/step - loss:
5.1769 - mae: 1.7396 - val_loss: 12.7382 - val_mae: 2.7731
Epoch 10/10
819/819 [==============================] - 42s 50ms/step - loss:
4.8313 - mae: 1.6762 - val_loss: 12.3602 - val_mae: 2.7286
405/405 [==============================] - 14s 32ms/step - loss:
10.0241 - mae: 2.4645
Test MAE: 2.46
```

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

Training and evaluating a dropout-regularized LSTM

- This model comprises two LSTM (Long Short-Term Memory) layers. The first layer has 64 units, followed by a second layer with 32 units.
- A dropout layer with a dropout rate of 0.4 is inserted between the two LSTM layers. Dropout is effective for regularizing the model and reducing overfitting by randomly dropping 40% of the units during training.

- The model is compiled using the RMSprop optimizer, a robust optimizer for training recurrent neural networks.
- Mean Squared Error (MSE) is chosen as the loss function to measure the difference between predicted and actual values.
- Mean Absolute Error (MAE) is selected as a metric to monitor during training, providing insight into the model's performance on the validation set.

```
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 [==============================] - 47s 53ms/step - loss:
23.9610 - mae: 3.5414 - val_loss: 9.8855 - val_mae: 2.4541
Epoch 2/10
819/819 [==============================] - 42s 51ms/step - loss:
10.1348 - mae: 2.4646 - val_loss: 11.0749 - val_mae: 2.6005
Epoch 3/10
819/819 [==============================] - 41s 50ms/step - loss:
8.1568 - mae: 2.2005 - val_loss: 11.5984 - val_mae: 2.6732
Epoch 4/10
819/819 [==============================] - 42s 51ms/step - loss:
6.8626 - mae: 2.0045 - val_loss: 12.2742 - val_mae: 2.7317
Epoch 5/10
819/819 [==============================] - 42s 51ms/step - loss:
6.0171 - mae: 1.8616 - val_loss: 12.3513 - val_mae: 2.7363
Epoch 6/10
819/819 [==============================] - 45s 54ms/step - loss:
5.3748 - mae: 1.7543 - val_loss: 13.1673 - val_mae: 2.8305
Epoch 7/10
819/819 [==============================] - 43s 53ms/step - loss:
4.9863 - mae: 1.6783 - val_loss: 13.1766 - val_mae: 2.8355
Epoch 8/10
819/819 [==============================] - 44s 54ms/step - loss:
4.5816 - mae: 1.6045 - val_loss: 13.5265 - val_mae: 2.8546
```

```
Epoch 9/10
819/819 [==============================] - 44s 54ms/step - loss:
4.2724 - mae: 1.5488 - val_loss: 13.2872 - val_mae: 2.8359
Epoch 10/10
819/819 [==============================] - 45s 54ms/step - loss:
4.0366 - mae: 1.5008 - val_loss: 13.8538 - val_mae: 2.8975
405/405 [==============================] - 15s 33ms/step - loss:
11.0696 - mae: 2.5973
Test MAE: 2.60
```

*With LSTM, the test MAE is 2.61 which is little similar to GRU.

*Both LSTM and GRU performed similarly with slight changes.

Using bidirectional RNNs

Training and evaluating a bidirectional LSTM

```
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:
27.4082 - mae: 3.7518 - val_loss: 10.5913 - val_mae: 2.5178
Epoch 2/10
819/819 [==============================] - 43s 53ms/step - loss:
9.3242 - mae: 2.3841 - val_loss: 10.2360 - val_mae: 2.4804
Epoch 3/10
819/819 [==============================] - 41s 50ms/step - loss:
8.3019 - mae: 2.2492 - val_loss: 10.8867 - val_mae: 2.5664
Epoch 4/10
819/819 [==============================] - 42s 51ms/step - loss:
7.6833 - mae: 2.1607 - val_loss: 10.9159 - val_mae: 2.5792
Epoch 5/10
819/819 [==============================] - 42s 51ms/step - loss:
7.1973 - mae: 2.0886 - val_loss: 10.7315 - val_mae: 2.5552
Epoch 6/10
819/819 [==============================] - 41s 49ms/step - loss:
6.8541 - mae: 2.0365 - val_loss: 10.9520 - val_mae: 2.5670
Epoch 7/10
819/819 [==============================] - 42s 50ms/step - loss:
6.5660 - mae: 1.9925 - val_loss: 11.0841 - val_mae: 2.6060
Epoch 8/10
```

```
819/819 [==============================] - 41s 50ms/step - loss:
6.3288 - mae: 1.9573 - val_loss: 11.4657 - val_mae: 2.6466
Epoch 9/10
819/819 [==============================] - 43s 52ms/step - loss:
6.1047 - mae: 1.9214 - val_loss: 11.7098 - val_mae: 2.6724
Epoch 10/10
819/819 [==============================] - 43s 52ms/step - loss:
5.9123 - mae: 1.8909 - val_loss: 11.9120 - val_mae: 2.6950
```

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 [==============================] - 45s 50ms/step - loss:
31.5131 - mae: 4.1944 - val_loss: 13.4678 - val_mae: 2.8615
Epoch 2/10
819/819 [==============================] - 46s 56ms/step - loss:
18.1330 - mae: 3.2976 - val_loss: 13.1475 - val_mae: 2.8610
Epoch 3/10
819/819 [==============================] - 49s 59ms/step - loss:
16.6743 - mae: 3.1564 - val_loss: 12.1540 - val_mae: 2.7516
Epoch 4/10
819/819 [==============================] - 38s 46ms/step - loss:
15.5995 - mae: 3.0509 - val_loss: 13.2649 - val_mae: 2.8615
Epoch 5/10
```

```
819/819 [==============================] - 38s 46ms/step - loss:
14.8402 - mae: 2.9686 - val_loss: 14.0597 - val_mae: 2.9364
Epoch 6/10
819/819 [==============================] - 38s 46ms/step - loss:
14.1350 - mae: 2.8923 - val_loss: 12.8393 - val_mae: 2.8183
Epoch 7/10
819/819 [==============================] - 38s 46ms/step - loss:
13.4870 - mae: 2.8195 - val_loss: 13.0671 - val_mae: 2.8414
Epoch 8/10
819/819 [==============================] - 37s 45ms/step - loss:
13.0179 - mae: 2.7634 - val_loss: 14.3496 - val_mae: 2.9776
Epoch 9/10
819/819 [==============================] - 47s 57ms/step - loss:
12.5907 - mae: 2.7154 - val_loss: 14.8073 - val_mae: 3.0079
Epoch 10/10
819/819 [==============================] - 38s 46ms/step - loss:
12.0859 - mae: 2.6578 - val_loss: 14.5226 - val_mae: 2.9854

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

405/405 [==============================] - 13s 29ms/step - loss:
14.5108 - mae: 3.0254
Test MAE: 3.03
```

The model deteriorated with test MAE 2.85 when conv1d and RNN lstm were combined.

Summary

I've created and evaluated a number of neural network designs for time series forecasting using the Jena Climate dataset as a case study. Using past climate data, the primary objective is to predict future temperature values with efficiency. Importing the dataset, which comprises meteorological observations for Jena, Germany, from 2009 to 2016, is the initial step. A thorough analysis and visualisation of the temperature time series are done to provide some initial findings. Primarily, the dataset is partitioned into training, validation, and test sets to provide dependable model assessment and prevent overfitting. To establish a baseline for comparison, a common sense approach is employed. The mean of the training data is used to forecast the temperature, and the resultant mean absolute error (MAE) is obtained.

This baseline is simple enough to serve as a benchmark for assessing the efficacy of more intricate models.

Densely Connected Model: AA basic model with two thick layers and an input layer of thirty-two units. The mean squared error (MSE) loss function and the RMSprop optimizer are employed. Despite being straightforward, this model achieves a respectable test MAE of 2.59.

1D Convolutional Model: Three 1D convolutional layers and max-pooling layers make up this model, which makes use of the capabilities of convolutional neural networks (CNNs). With a higher test MAE of 3.20, it compares badly to the densely connected model.

RNNs, or recurrent neural networks: Given that time series data are sequential, a variety of RNN topologies are examined, including a simple RNN model that utilizes Keras' SimpleRNN layer and has stacked layers and increasing units. The poor performance of the model is indicated by its high test MAE of 9.90.

stacked Gated Recurrent Unit (GRU) model with dropout regularisation to prevent overfitting. This model has an excellent test MAE of 2.47.

Long Short-Term Memory (LSTM) Layered model with dropout regularization integrated. With a test MAE of 2.61, it shows comparable performance to the GRU model.

Combination of 1D Convolution and RNN: To combine the benefits of both convolutional and recurrent layers, a hybrid model consisting of an LSTM layer, dropout regularisation, and a 1D convolutional layer is created. With a test MAE of 2.85, this model nevertheless performs worse than the stacked GRU and LSTM models.

The stacked GRU and LSTM models exhibit the lowest test MAE among all tested models, making them the top performing architectures. They are able to identify long-term dependencies in the time series data and the regularization dropout they provide is what accounts for their enhanced performance. This extensive investigation, taken as a whole, provides a systematic way to create and evaluate multiple neural network topologies for time series forecasting, using the Jena Climate dataset as a helpful case study. As compared to other architectures studied, the results show how successfully stacked GRU and LSTM models discover intricate patterns and linkages in the climatic data.