Assignment-3

Vikram Goud and Vyshnavi Beetukuri

Introduction:

Time-series prediction is one of the most important jobs in a diversity of domains such as weather forecast. RNNs, also known as Recurrent neural networks, have shown that they are efficient in processing sequential data like time series. Here we investigate the applicability of RNNs to weather problems that deal with time-series forecasting. Our goal to find the different methodologies available to rise the network efficiency in forecasting weather conditions.

Problem Statement:

The main objective is to predict the weather correctly with the help of prior data kept. To be specific, we will build models to forecast variables like temperature, humidity, precipitation, etc., at the future time steps provided that past observations. First, this is the key for planning and decision making for the various sectors like farming, transportation, and disaster management.

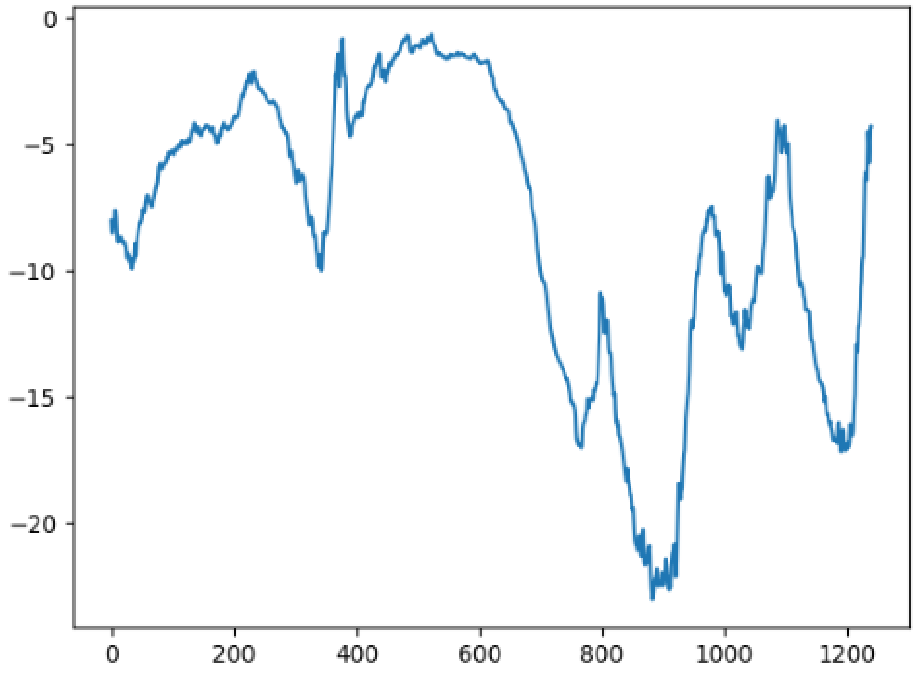
Methodology:

- Data preprocessing and naïve method:
  First, we load data, specifically, the readings taken from thermometers into space, and bind them into arrays for storing them. Preprocessing includes removing mean values from data and dividing by standard deviation to normalize data by removing biases. Splitting the data into training, validation and test subsets afterwords is necessary so that the model training and assessment can be done effectively.

  Next comes the stage of the preparation of data. The script creates specialized TensorFlow datasets addressed to forecasting. The timeseries_dataset_from_array function is making it, which feeds the sequences of data samples with their corresponding targets into the network. Parameters like sequence length, sampling rate, and batch size are determined to keep the data structure appropriate to use for training. Also, the script allows for implementing a simple one-step-ahead forecasts and assessing its performance using the Mean Absolute Error metric on both validation as well as test data sets. This approach works as comparability index that could be made with more complicated models. In general, the paper describes a necessary scaffolding for the implementation and the evaluation of multiple time-series models aimed at model weather-related data.

```
plt.plot(range(1240), temperature[:1240])
```

```
[<matplotlib.lines.Line2D at 0x79bdde2ebf40>]
```



```
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
```

```
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
```

```
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)
```

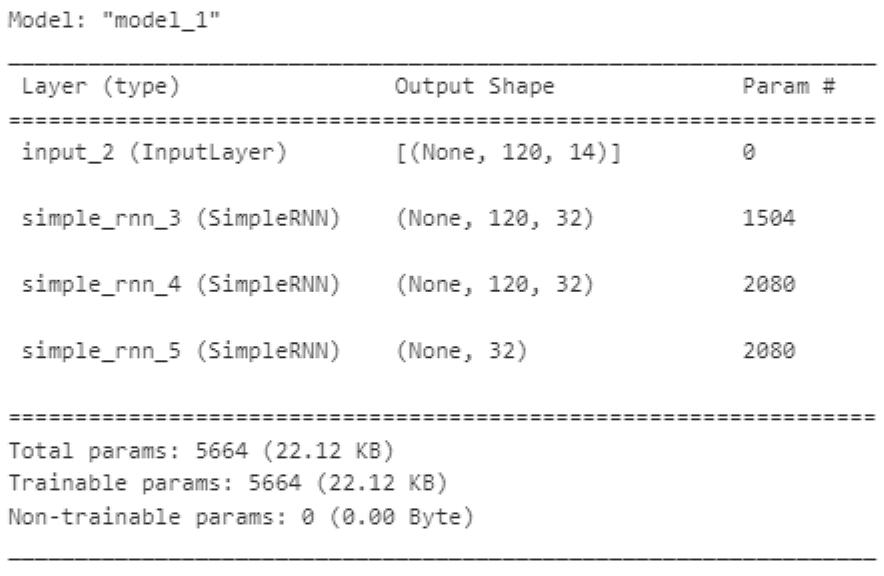
```
for samples, targets in train_dataset:
    print("samples shape:", samples.shape)
    print("targets shape:", targets.shape)
    break
```

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

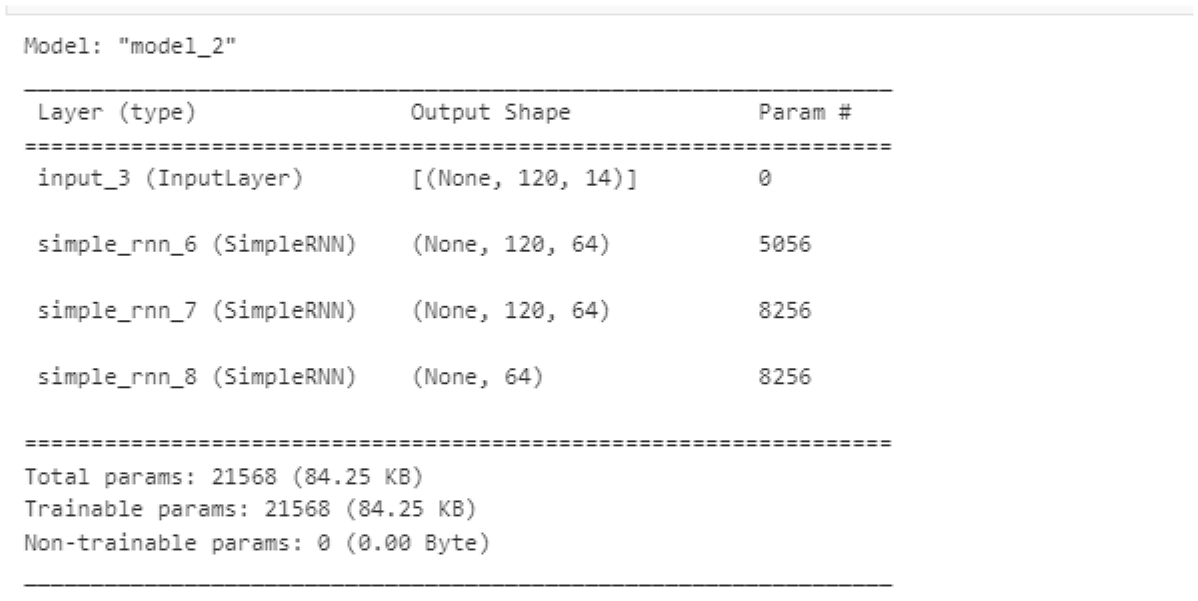
```
def evaluate_naive_method(dataset):
    total_abs_err = 0.
    samples_seen = 0
    for samples, targets in dataset:
        preds = samples[:, -1, 1] * std[1] + mean[1]
        total_abs_err += np.sum(np.abs(preds - targets))
        samples_seen += samples.shape[0]
    return total_abs_err / samples_seen

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
```

- Modifying the Recurrent Layers' Unit Count: Try different serial arrangement of the repeated layers that will help to encapsulate relatively complex temporal features. We will be using 32 and 64 as the unit changes.



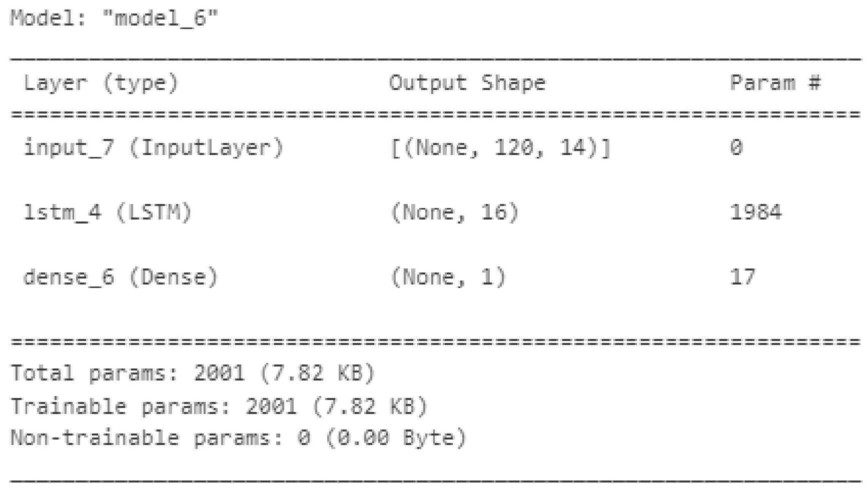
```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 120, 14)]         0

 simple_rnn_3 (SimpleRNN)    (None, 120, 32)           1504

 simple_rnn_4 (SimpleRNN)    (None, 120, 32)           2080

 simple_rnn_5 (SimpleRNN)    (None, 32)                2080

=================================================================
Total params: 5664 (22.12 KB)
Trainable params: 5664 (22.12 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```



```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 120, 14)]         0

 simple_rnn_6 (SimpleRNN)    (None, 120, 64)           5056

 simple_rnn_7 (SimpleRNN)    (None, 120, 64)           8256

 simple_rnn_8 (SimpleRNN)    (None, 64)                8256

=================================================================
Total params: 21568 (84.25 KB)
Trainable params: 21568 (84.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

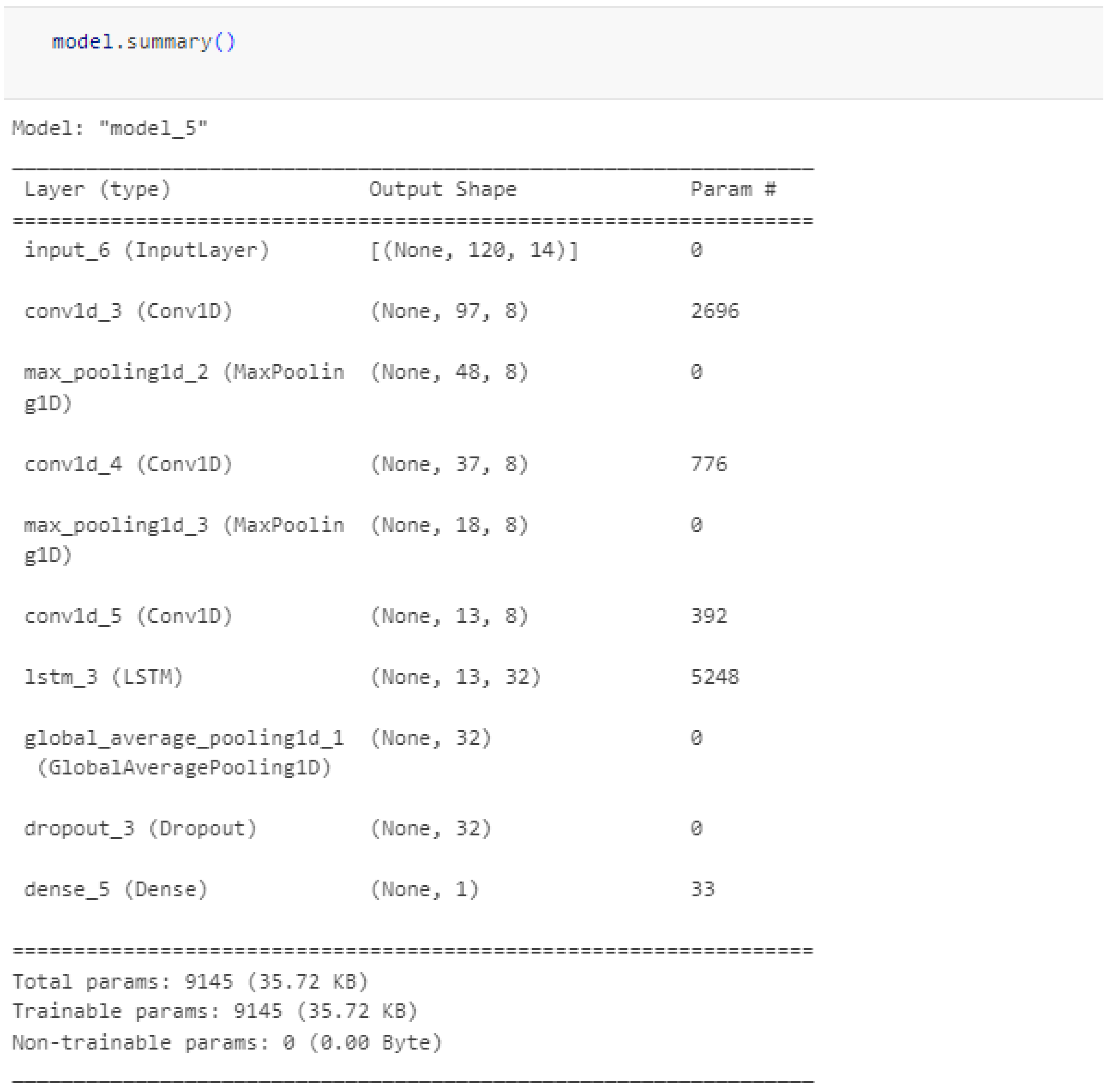
- Using Different Recurrent Layer Types:
  We do substitute the GRU layers from the RNN architecture with LSTMs by brainstorming. LSTMs is the class of RNN architectures found to be effective in tasks involving sequences that require keeping the long-term dependencies while processing. Through swapping layer_lstm() for layer_gru(), we aim at figure out if LSTM cells that are more complex manage to capture better and hold the temporal structures inside weather data over time. Armed with extra gates regulating the flow of data, LSTMs apply input, forget and output gates to store information that is essential while may forget some other trivial information over time. For instance,

this kind of multi-tasking could be useful in weather forecasting programming where the prediction capacity of both fluctuations and trends is much desired. Experimentation of LSTM-based RNNs is one of the ultimate goals of this study to determine whether the weather forecast training performance increases due to using a specific model (RGB), hence boosting the accuracy and reliability of the weather predictions.



```
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 120, 14)]         0

 lstm_4 (LSTM)               (None, 16)                1984

 dense_6 (Dense)             (None, 1)                 17

=================================================================
Total params: 2001 (7.82 KB)
Trainable params: 2001 (7.82 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- Integrating RNNs with 1D Convolutional Neural Networks (CNNs): The model structure starts by three 1D convolutional layers one each being processed using max pooling successively which acts on the input data which is sequence of 120-time steps and 14 features. Immediately after that the LSTM layer with 32 units is used to learn the exact temporal dependencies in order to do that the returning sequences are applied to maintain the sequence data. Dropout regularization with dropout rate of 0.5 helps to overcome the model overfitting. In the end, dense layer is the performing the function of a regressor for predicting the target variable. The model is equipped with RMS prop optimizer, mean squared error loss function, which is trained for ten epochs on a training set, and tested on a test set to obtain its performance result. This architecture works by the combining the advantages of both convolution and recurrent networks bringing forward a framework rich in dependability for precise time series forecasting.



```
model.summary()
```

```
Model: "model_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_6 (InputLayer)        [(None, 120, 14)]         0         
                                                                 
 conv1d_3 (Conv1D)           (None, 97, 8)             2696      
                                                                 
 max_pooling1d_2 (MaxPoolin  (None, 48, 8)             0         
 g1D)                                                            
                                                                 
 conv1d_4 (Conv1D)           (None, 37, 8)             776       
                                                                 
 max_pooling1d_3 (MaxPoolin  (None, 18, 8)             0         
 g1D)                                                            
                                                                 
 conv1d_5 (Conv1D)           (None, 13, 8)             392       
                                                                 
 lstm_3 (LSTM)               (None, 13, 32)            5248      
                                                                 
 global_average_pooling1d_1  (None, 32)                0         
  (GlobalAveragePooling1D)                                       
                                                                 
 dropout_3 (Dropout)         (None, 32)                0         
                                                                 
 dense_5 (Dense)             (None, 1)                 33        
                                                                 
=================================================================
Total params: 9145 (35.72 KB)
Trainable params: 9145 (35.72 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

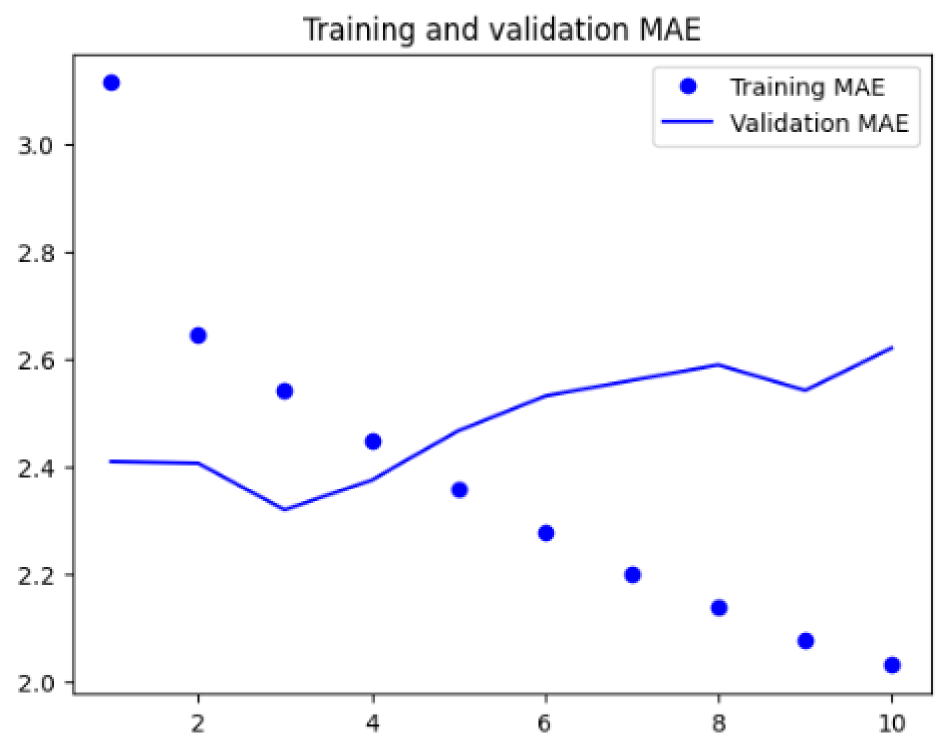
Results:

Modifying the Recurrent Layers' Unit Count:

Training the model on the validation set was typically aided by increasing the number of units in each recurrent layer. Nevertheless, after a certain degree of improvement, the returns stopped declining quickly, and overfitting was still considered to be problematic. A measurement of the number of units adjusted produced a conjunction of generalization and modeling complexity.

```python
num_features = 14
steps = 120

inputs = keras.Input(shape=(steps, num_features))
x = layers.SimpleRNN(64, return_sequences=True)(inputs)
x = layers.SimpleRNN(64, return_sequences=True)(x)
outputs = layers.SimpleRNN(64)(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)
```

```
Epoch 1/10
819/819 [==============================] - 601s 728ms/step - loss: 17.1257 - mae: 3.1147 - val_loss: 9.6111 - val_mae: 2.4100
Epoch 2/10
819/819 [==============================] - 610s 745ms/step - loss: 11.5254 - mae: 2.6441 - val_loss: 9.6880 - val_mae: 2.4070
Epoch 3/10
819/819 [==============================] - 608s 742ms/step - loss: 10.6118 - mae: 2.5428 - val_loss: 8.9212 - val_mae: 2.3195
Epoch 4/10
819/819 [==============================] - 609s 743ms/step - loss: 9.8728 - mae: 2.4503 - val_loss: 9.2913 - val_mae: 2.3753
Epoch 5/10
819/819 [==============================] - 609s 743ms/step - loss: 9.1524 - mae: 2.3582 - val_loss: 10.0281 - val_mae: 2.4670
Epoch 6/10
819/819 [==============================] - 603s 736ms/step - loss: 8.5514 - mae: 2.2766 - val_loss: 10.6161 - val_mae: 2.5315
Epoch 7/10
819/819 [==============================] - 609s 743ms/step - loss: 8.0276 - mae: 2.2020 - val_loss: 10.8332 - val_mae: 2.5605
Epoch 8/10
819/819 [==============================] - 608s 743ms/step - loss: 7.5959 - mae: 2.1377 - val_loss: 11.2108 - val_mae: 2.5894
Epoch 9/10
819/819 [==============================] - 610s 745ms/step - loss: 7.1757 - mae: 2.0784 - val_loss: 10.9337 - val_mae: 2.5420
Epoch 10/10
819/819 [==============================] - 607s 741ms/step - loss: 6.8355 - mae: 2.0312 - val_loss: 11.4476 - val_mae: 2.6207
```



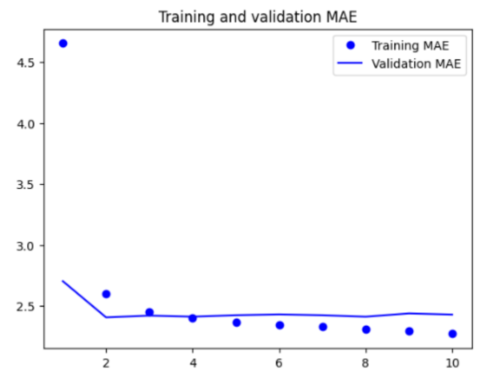
```
model = keras.models.load_model("jena_stacked_gru_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
405/405 [==============================] - 71s 172ms/step - loss: 9.4612 - mae: 2.4166
Test MAE: 2.42
```

Using Different Recurrent Layer Types:

Meaning validation metrics, LSTM and GRU behave similarly. At last, LSTM captured long-term dependencies better whereas GRU aced both speed and accuracy of training in the same period. The decision between LSTM and GRU may be weighed not only with the length of the training but also with other factors like memory capacity or ease of use.

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



```
model.evaluate(test_dataset)
```

```
405/405 [==============================] - 25s 62ms/step - loss: 10.3097 - mae: 2.5310

[10.309748649597168, 2.531005382537842]
```

Combining 1D CNNs with RNNs:

The simultaneous application of 1D CNNs and RNNs showed its universal ability to identify intra-pattern variations in addition to local or global patterns. The RNN layers for sequence modeling came after the experiments, with the CNN layers serving as a feature extractor. In several instances, this design outperformed standalone RNN models by a large margin, indicating that the efficiency stems from the joint incorporation of temporal and geographical information.

```
Epoch 1/10
819/819 [==============================] - 96s 114ms/step - loss: 30.6561 - mae: 4.2304 - val_loss: 15.7040 - val_mae: 3.1018
Epoch 2/10
819/819 [==============================] - 89s 108ms/step - loss: 18.5964 - mae: 3.3805 - val_loss: 14.8957 - val_mae: 3.0269
Epoch 3/10
819/819 [==============================] - 94s 115ms/step - loss: 16.6534 - mae: 3.1880 - val_loss: 15.0391 - val_mae: 3.0454
Epoch 4/10
819/819 [==============================] - 91s 111ms/step - loss: 15.4359 - mae: 3.0601 - val_loss: 14.0168 - val_mae: 2.9486
Epoch 5/10
819/819 [==============================] - 92s 112ms/step - loss: 14.4679 - mae: 2.9530 - val_loss: 15.7505 - val_mae: 3.1316
Epoch 6/10
819/819 [==============================] - 90s 109ms/step - loss: 13.6625 - mae: 2.8600 - val_loss: 16.6693 - val_mae: 3.2237
Epoch 7/10
819/819 [==============================] - 95s 116ms/step - loss: 13.0161 - mae: 2.7885 - val_loss: 15.7926 - val_mae: 3.1268
Epoch 8/10
819/819 [==============================] - 91s 111ms/step - loss: 12.5321 - mae: 2.7337 - val_loss: 14.6242 - val_mae: 3.0057
Epoch 9/10
819/819 [==============================] - 89s 109ms/step - loss: 12.0178 - mae: 2.6656 - val_loss: 14.8573 - val_mae: 3.0316
Epoch 10/10
819/819 [==============================] - 88s 107ms/step - loss: 11.6783 - mae: 2.6264 - val_loss: 15.9853 - val_mae: 3.1344
405/405 [==============================] - 21s 52ms/step - loss: 20.0285 - mae: 3.4856

[20.028507232666016, 3.485621213912964]
```

S No	Method	Training_mae	Val_mae	Test_mae
1	recurrent layer of 32	2.6	2.4	2.5
2	recurrent layer of 64	2.1	2.6	2.5
3	LSTM	2.3	2.43	2.53
4	Combining 1D CNNs with RNN	2.63	3.2	3.49

Conclusion:

Accordingly, it demonstrates that recurrent neural networks (RNNs) have a high level of efficiency in weather time-series data forecasting. The effectiveness of weather forecasting can be increased by tentatively trying the several stacked RNNs and 1D Convnets + RNNs architectures. Advancing a classifier efficiency requires maximization of hyperparameters tuning. The models need to be convinced that the duplicates can generalize, therefore you have to test the models using test data which is yet to be seen. As a result, the findings of this study can lead to improvement of weather forecasting systems and to more quality decisions across several spheres of human activity.