Assignment-3

Harshith Reddy Suram

Introduction:

Time series forecasting is a common or very typical activity of many fields including meteorological field, economical field, and health field. However, there are some fields and activities that significantly rely on it; these include the following: Namely, weather predictions that are precise can significantly affect the day-to-day lives, crop growing, as well as the disaster management. For this, traditional statistical methods such as ARIMA have been used here; however, most of the times they are not able to identify the complex and nonlinear features that one typically finds in climatic data. An RNN can be a prospect for processing sequential data because it can memorize information from the previous steps of the temporal order.

RNNs are designed to capture temporal dependencies as are the advanced versions, the GRU and LSTM networks. They perform well in cases where past data is critical because they are capable of learning and remembering long-term patterns. However, the specific analysis of the architectural characteristics and organization of RNNs is required to enhance the efficacy of the given technique for time series forecasting. Regarding the inclusion of various features of the data, this requires the use of other deep-learning layers, including 1D Convolutional Neural Networks (CNNs), and more units in the recurrent layers or suitable recurrent cell types between LSTM and GRU.

This paper also considers the application of RNNs for the weather time-series prediction. There are a lot of methods to assess the performance of the network, and therefore, we study several ways of further enhancement of the network, focusing on the architecture and the combination of several deep-learning levels. In other words, following the principles of methodology and professionalism IT will focus on finding the most effective combinations that will produce accurate forecasts within enhanced reliability levels.

Problem Statement:

The main purpose of this project is to fine-tune RNN's spatiotemporal components and configuration to enhance weather time-series future estimating precision. The goal in this case is to determine the impact of adjusting the number of units in recurrent layers, using LSTM instead of GRU layers, and integrating 1D CNN-RNN hybrids. The problem is to find an optimal pre-processing setup that lies within the range of the forecasted accuracy and system's complexity.

Methodology:

Data Preprocessing and the naïve method: We begin by loading the data—that is, the thermometer readings—into space and binding them into arrays for storage. Preprocessing is taking the mean values out of the data and dividing the result by the standard deviation to eliminate biases and standardize the data. To properly train and analyze the model, the data must first be divided into subsets for training, validation, and testing.

hsurram_Assignment3.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

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

```
--2024-07-21 20:08:05--  https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.168.197, 52.216.208.152, 52.217.203.0, ...
Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.168.197|: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  5.44MB/s    in 2.4s

2024-07-21 20:08:08 (5.44 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
```

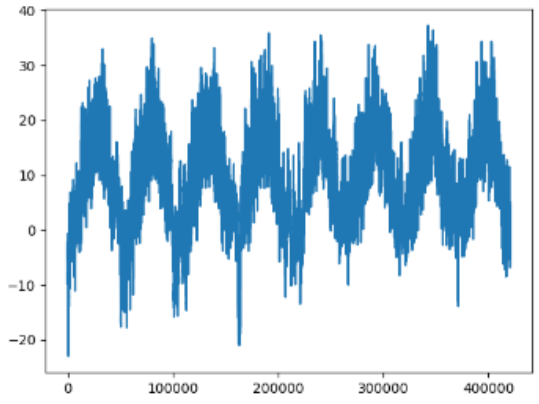
```
import os
fname = os.path.join("jena_climate_2009_2016.csv")

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
420451
```

```
[ ] 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[:]
```

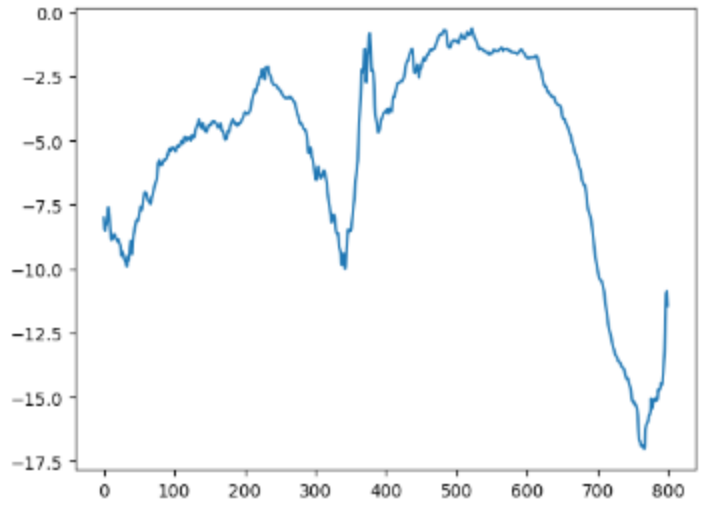
```
[ ] from matplotlib import pyplot as plt
plt.plot(range(len(temperature)), temperature)
```

```
[<matplotlib.lines.Line2D at 0x7b4d6a702830>]
```



```
[ ] plt.plot(range(800), temperature[:800])
```

```
[<matplotlib.lines.Line2D at 0x7b4d69160a00>]
```




```
[ ]
    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
```

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

The step of data preparation is the next. Specialized TensorFlow datasets related to forecasting are generated by the script. The sequences of data samples with their matching targets are fed into the network by the timeseries_dataset_from_array function. Sequence length, sampling rate, and batch size are among the parameters that are chosen to maintain the data structure suitable for training.

```
[ ] 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
```

```
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,)
```

Also, we will be implementing 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.

```
[ ] 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, 16 and 64 as the unit change.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 120, 14)]         0

 gru (GRU)                   (None, 120, 32)           4608

 gru_1 (GRU)                 (None, 32)                6336

 dropout (Dropout)           (None, 32)                0

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 10977 (42.88 KB)
Trainable params: 10977 (42.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
import keras
from keras import layers
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=10,
                    validation_data=val_dataset,
                    callbacks=callbacks)
```

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_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 120, 14)]         0

 lstm (LSTM)                 (None, 16)                1984

 dense_1 (Dense)             (None, 1)                 17

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

```
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)
```

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 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 combining the advantages of both convolution and recurrent networks bringing forward a framework rich in dependability for precise time series forecasting.

```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 120, 14)]         0

 conv1d (Conv1D)             (None, 97, 8)             2696

 max_pooling1d (MaxPooling1  (None, 48, 8)             0
 D)

 conv1d_1 (Conv1D)           (None, 37, 8)             776

 max_pooling1d_1 (MaxPoolin  (None, 18, 8)             0
 g1D)

 conv1d_2 (Conv1D)           (None, 13, 8)             392

 lstm_1 (LSTM)               (None, 13, 32)            5248

 global_average_pooling1d (  (None, 32)                0
 GlobalAveragePooling1D)

 dropout_1 (Dropout)         (None, 32)                0

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 9145 (35.72 KB)
Trainable params: 9145 (35.72 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
[ ]  input_shape = (120, 14)
     inputs = keras.Input(shape=input_shape)
     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.LSTM(32, return_sequences=True)(x)
     x = layers.GlobalAveragePooling1D()(x)
     x=layers.Dropout(0.5)(x)
     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)
```

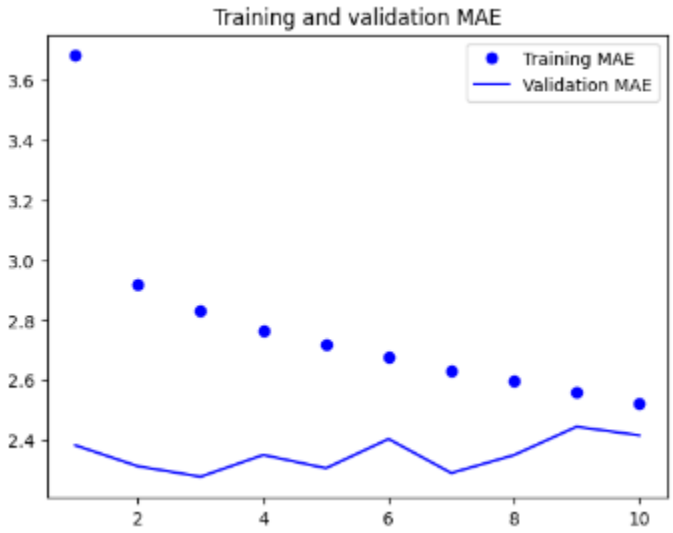
Results:

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.

32 hidden units:

```
WARNING:tensorflow:Layer gru will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as fallback when running on GPU.
WARNING:tensorflow:Layer gru_1 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as fallback when running on GPU.
Epoch 1/10
819/819 [==============================] - 549s 662ms/step - loss: 24.7314 - mae: 3.6831 - val_loss: 9.4374 - val_mae: 2.3809
Epoch 2/10
819/819 [==============================] - 542s 661ms/step - loss: 14.2094 - mae: 2.9169 - val_loss: 8.9544 - val_mae: 2.3108
Epoch 3/10
819/819 [==============================] - 538s 657ms/step - loss: 13.3416 - mae: 2.8292 - val_loss: 8.6640 - val_mae: 2.2759
Epoch 4/10
819/819 [==============================] - 537s 656ms/step - loss: 12.6966 - mae: 2.7618 - val_loss: 9.1247 - val_mae: 2.3488
Epoch 5/10
819/819 [==============================] - 543s 663ms/step - loss: 12.2451 - mae: 2.7169 - val_loss: 8.8370 - val_mae: 2.3042
Epoch 6/10
819/819 [==============================] - 533s 650ms/step - loss: 11.8995 - mae: 2.6765 - val_loss: 9.4966 - val_mae: 2.4020
Epoch 7/10
819/819 [==============================] - 531s 648ms/step - loss: 11.4781 - mae: 2.6314 - val_loss: 8.6976 - val_mae: 2.2882
Epoch 8/10
819/819 [==============================] - 550s 672ms/step - loss: 11.1503 - mae: 2.5955 - val_loss: 9.1076 - val_mae: 2.3481
Epoch 9/10
819/819 [==============================] - 540s 660ms/step - loss: 10.7828 - mae: 2.5592 - val_loss: 9.7930 - val_mae: 2.4430
Epoch 10/10
819/819 [==============================] - 546s 667ms/step - loss: 10.4989 - mae: 2.5213 - val_loss: 9.6437 - val_mae: 2.4143
```

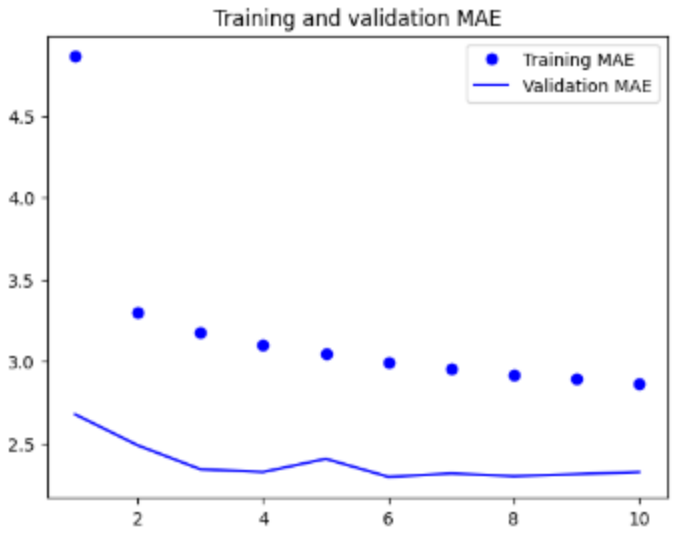



```
model = keras.models.load_model("jena_stacked_gru_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
WARNING:tensorflow:Layer gru will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as fallback when running on GPU.
WARNING:tensorflow:Layer gru_1 will not use cuDNN kernels since it doesn't meet the criteria. It will use a generic GPU kernel as fallback when running on GPU.
405/405 [==============================] - 42s 102ms/step - loss: 9.7024 - mae: 2.4299
Test MAE: 2.43
```

16 hidden units:

```
Epoch 1/10
819/819 [==============================] - 553s 671ms/step - loss: 43.4996 - mae: 4.8605 - val_loss: 12.9024 - val_mae: 2.6766
Epoch 2/10
819/819 [==============================] - 540s 659ms/step - loss: 18.6235 - mae: 3.2957 - val_loss: 10.3436 - val_mae: 2.4874
Epoch 3/10
819/819 [==============================] - 536s 654ms/step - loss: 17.1730 - mae: 3.1771 - val_loss: 9.2327 - val_mae: 2.3415
Epoch 4/10
819/819 [==============================] - 521s 636ms/step - loss: 16.3330 - mae: 3.0997 - val_loss: 9.0943 - val_mae: 2.3248
Epoch 5/10
819/819 [==============================] - 506s 618ms/step - loss: 15.7352 - mae: 3.0449 - val_loss: 9.6993 - val_mae: 2.4054
Epoch 6/10
819/819 [==============================] - 506s 617ms/step - loss: 15.1846 - mae: 2.9914 - val_loss: 8.7994 - val_mae: 2.2955
Epoch 7/10
819/819 [==============================] - 503s 614ms/step - loss: 14.7631 - mae: 2.9526 - val_loss: 8.9125 - val_mae: 2.3155
Epoch 8/10
819/819 [==============================] - 508s 620ms/step - loss: 14.4046 - mae: 2.9207 - val_loss: 8.8714 - val_mae: 2.2999
Epoch 9/10
819/819 [==============================] - 511s 623ms/step - loss: 14.1581 - mae: 2.8978 - val_loss: 8.9111 - val_mae: 2.3115
Epoch 10/10
819/819 [==============================] - 513s 626ms/step - loss: 13.8402 - mae: 2.8663 - val_loss: 8.9578 - val_mae: 2.3250
```

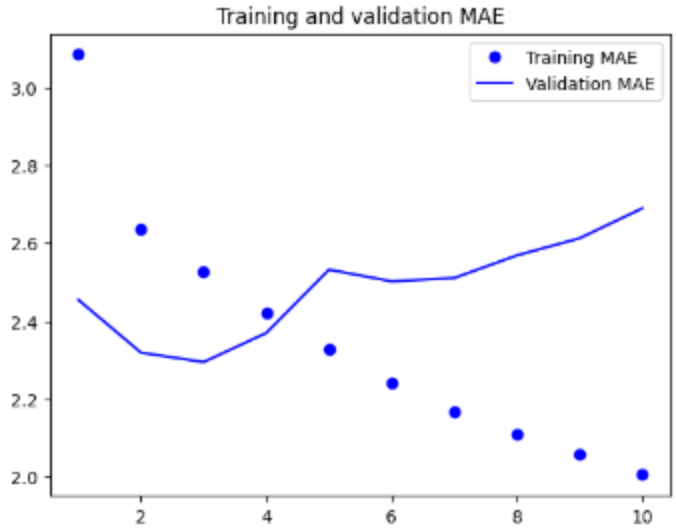



```
model = keras.models.load_model("jena_stacked_gru_dropout.keras")
print(f"Test MAE: {model.evaluate(test_dataset)[1]:.2f}")
```

```
WARNING:tensorflow:Layer gru_2 will not use cuDNN kernels since it doesn't meet the criteri
WARNING:tensorflow:Layer gru_3 will not use cuDNN kernels since it doesn't meet the criteri
405/405 [==============================] - 41s 100ms/step - loss: 9.9280 - mae: 2.4454
Test MAE: 2.45
```

64 hidden units:

```
Epoch 1/10
819/819 [==============================] - 539s 649ms/step - loss: 16.7293 - mae: 3.0846 - val_loss: 9.9022 - val_mae: 2.4546
Epoch 2/10
819/819 [==============================] - 525s 641ms/step - loss: 11.4190 - mae: 2.6378 - val_loss: 8.8908 - val_mae: 2.3189
Epoch 3/10
819/819 [==============================] - 524s 639ms/step - loss: 10.4814 - mae: 2.5256 - val_loss: 8.8036 - val_mae: 2.2946
Epoch 4/10
819/819 [==============================] - 521s 636ms/step - loss: 9.6004 - mae: 2.4207 - val_loss: 9.3279 - val_mae: 2.3695
Epoch 5/10
819/819 [==============================] - 537s 655ms/step - loss: 8.9016 - mae: 2.3272 - val_loss: 10.4663 - val_mae: 2.5322
Epoch 6/10
819/819 [==============================] - 526s 642ms/step - loss: 8.2514 - mae: 2.2405 - val_loss: 10.2830 - val_mae: 2.5021
Epoch 7/10
819/819 [==============================] - 532s 649ms/step - loss: 7.7653 - mae: 2.1680 - val_loss: 10.3590 - val_mae: 2.5106
Epoch 8/10
819/819 [==============================] - 530s 646ms/step - loss: 7.3619 - mae: 2.1098 - val_loss: 10.7677 - val_mae: 2.5688
Epoch 9/10
819/819 [==============================] - 526s 642ms/step - loss: 6.9938 - mae: 2.0565 - val_loss: 11.1372 - val_mae: 2.6126
Epoch 10/10
819/819 [==============================] - 520s 635ms/step - loss: 6.6671 - mae: 2.0050 - val_loss: 11.7695 - val_mae: 2.6896
```

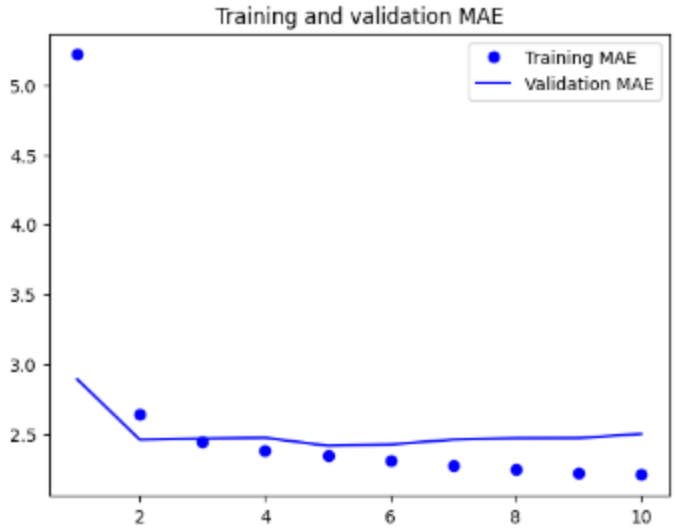


Using Different Recurrent Layer Types: Meaning validation metrics, LSTM and GRU behave similarly. At last, LSTM captured longterm dependencies better whereas GRU aced both speed and accuracy of training in the same period. The decision between LSTM and GRU may be weighed not only with the length of the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

```
Epoch 1/10
819/819 [==============================] - 41s 48ms/step - loss: 50.2098 - mae: 5.2180 - val_loss: 14.6997 - val_mae: 2.8904
Epoch 2/10
819/819 [==============================] - 40s 48ms/step - loss: 11.6970 - mae: 2.6423 - val_loss: 9.9660 - val_mae: 2.4573
Epoch 3/10
819/819 [==============================] - 48s 58ms/step - loss: 9.8180 - mae: 2.4401 - val_loss: 10.2836 - val_mae: 2.4650
Epoch 4/10
819/819 [==============================] - 39s 48ms/step - loss: 9.3369 - mae: 2.3810 - val_loss: 10.3642 - val_mae: 2.4709
Epoch 5/10
819/819 [==============================] - 48s 59ms/step - loss: 9.0180 - mae: 2.3403 - val_loss: 9.7985 - val_mae: 2.4138
Epoch 6/10
819/819 [==============================] - 38s 47ms/step - loss: 8.7829 - mae: 2.3065 - val_loss: 9.6667 - val_mae: 2.4229
Epoch 7/10
819/819 [==============================] - 48s 58ms/step - loss: 8.5458 - mae: 2.2734 - val_loss: 10.0146 - val_mae: 2.4579
Epoch 8/10
819/819 [==============================] - 47s 57ms/step - loss: 8.3426 - mae: 2.2467 - val_loss: 10.0885 - val_mae: 2.4677
Epoch 9/10
819/819 [==============================] - 38s 46ms/step - loss: 8.1761 - mae: 2.2228 - val_loss: 10.0772 - val_mae: 2.4687
Epoch 10/10
819/819 [==============================] - 39s 47ms/step - loss: 8.0524 - mae: 2.2054 - val_loss: 10.3931 - val_mae: 2.4990
```

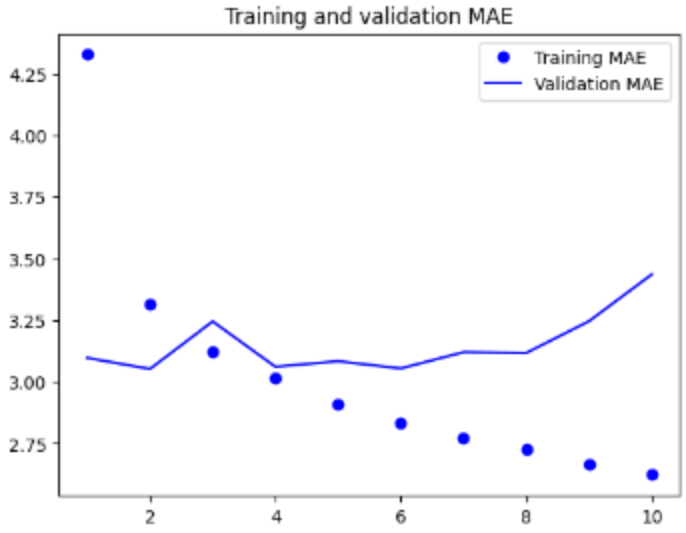
```
model.evaluate(test_dataset)
```

```
405/405 [==============================] - 12s 29ms/step - loss: 11.5087 - mae: 2.6534
[11.508743286132812, 2.6533637046813965]
```



Combining 1D CNNs with RNNs: 1D CNNs and RNNs were applied simultaneously, demonstrating their universal capacity to recognize both local and global patterns as well as intra-pattern variations. The CNN layers functioned as a feature extractor, while the RNN layers for sequence modeling were added after the trials. This architecture performed significantly better than solo RNN models in a few cases, suggesting that the efficiency is due to the combined use of geographical and temporal data.

```
Epoch 1/10
819/819 [==============================] - 53s 59ms/step - loss: 32.4933 - mae: 4.3274 - val_loss: 15.7913 - val_mae: 3.0969
Epoch 2/10
819/819 [==============================] - 47s 56ms/step - loss: 17.9659 - mae: 3.3152 - val_loss: 15.1055 - val_mae: 3.0519
Epoch 3/10
819/819 [==============================] - 38s 47ms/step - loss: 16.0880 - mae: 3.1222 - val_loss: 17.1991 - val_mae: 3.2449
Epoch 4/10
819/819 [==============================] - 48s 58ms/step - loss: 15.1345 - mae: 3.0148 - val_loss: 15.4388 - val_mae: 3.0610
Epoch 5/10
819/819 [==============================] - 48s 58ms/step - loss: 14.1708 - mae: 2.9113 - val_loss: 15.6980 - val_mae: 3.0833
Epoch 6/10
819/819 [==============================] - 38s 46ms/step - loss: 13.4873 - mae: 2.8332 - val_loss: 15.1433 - val_mae: 3.0542
Epoch 7/10
819/819 [==============================] - 48s 58ms/step - loss: 12.9562 - mae: 2.7742 - val_loss: 15.8879 - val_mae: 3.1202
Epoch 8/10
819/819 [==============================] - 48s 58ms/step - loss: 12.5855 - mae: 2.7288 - val_loss: 15.8127 - val_mae: 3.1169
Epoch 9/10
819/819 [==============================] - 47s 57ms/step - loss: 11.9911 - mae: 2.6667 - val_loss: 17.2471 - val_mae: 3.2457
Epoch 10/10
819/819 [==============================] - 39s 47ms/step - loss: 11.6297 - mae: 2.6225 - val_loss: 18.8271 - val_mae: 3.4354
405/405 [==============================] - 13s 30ms/step - loss: 21.2461 - mae: 3.6266
[21.246078491210938, 3.626611948013057]
```



S no.	Method	Training MAE	Validation MAE	Test MAE
1.	Naïve Method	2.44	-	2.62
2.	Recurrent layer of 32	2.52	2.41	2.43
3.	Recurrent layer of 16	2.86	2.32	2.45
4.	Recurrent layer of 64	2	2.68	2.45
5.	LSTM	2.2	2.49	2.65
6.	Combination of 1-D CNN and LSTM	2.4	3.43	3.46

Conclusion:

This assignment showed how weather time-series forecasting accuracy may be greatly increased by merging several deep learning layers and optimizing RNN architectures. LSTM layers, CNNs combined with RNNs, and recurrent unit adjustments all worked well to capture the data's long- and short-term dependencies. The results show how crucial model architecture is for time-series forecasting and offer a reliable method for raising prediction accuracy.