Final Exam

MATH 686

James Della-Giustina

Due by midnight of December 15th, 2022

Part (I) 30 Points

Run the LSTM model for Jena temperature forecasting.

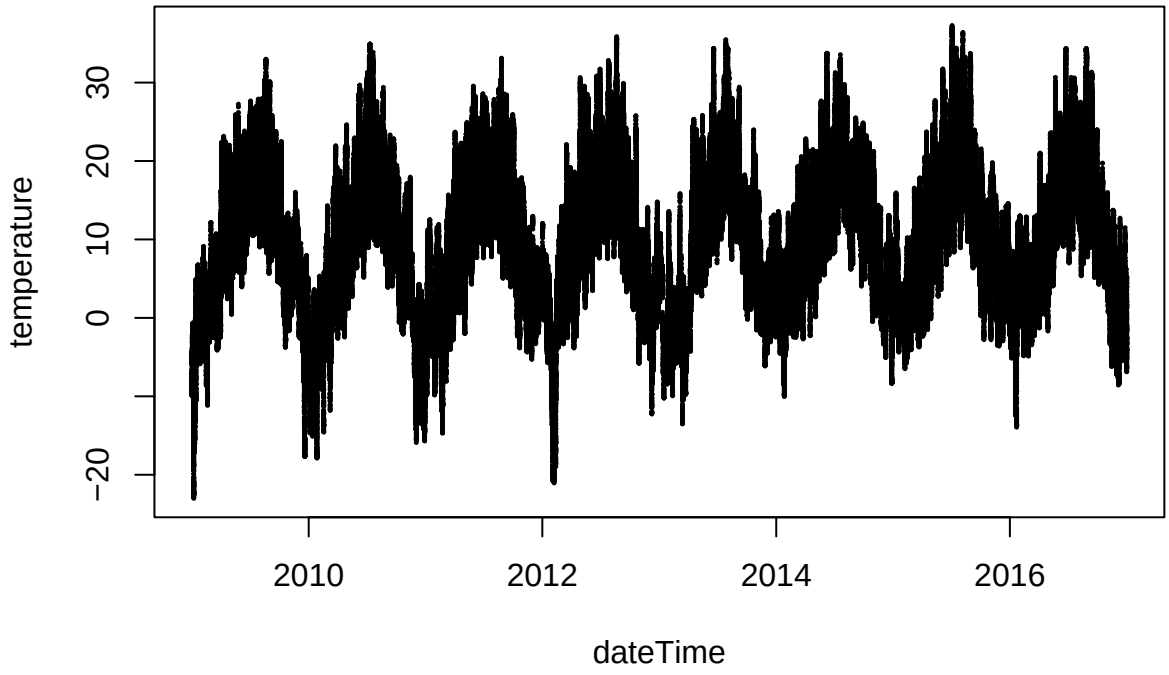
- 1. Find out the MAE for the testing data.
- 2. Find out the MSE for the testing data.

```
library(magrittr)
```

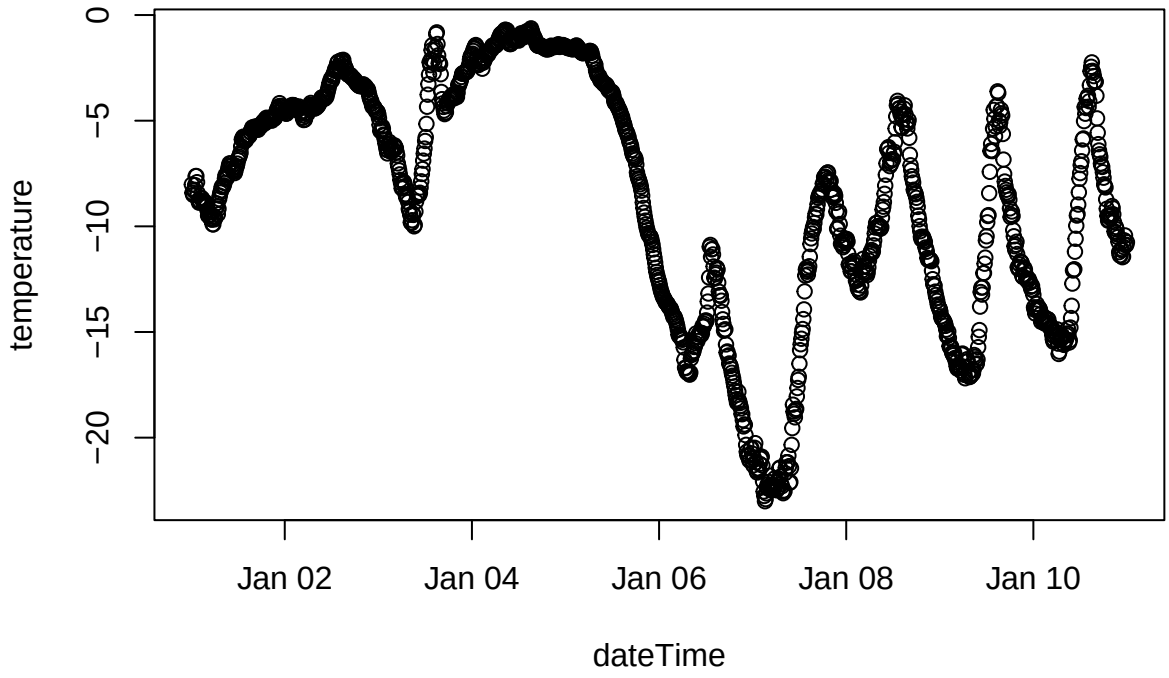
```
## Warning: package 'magrittr' was built under R version 4.1.2
```

```
full_df$`Date Time` %<>% as.POSIXct(tz = "Etc/GMT+1", format = "%d.%m.%Y %H:%M:%S")
# x %<>% fn()  is shorthand for x <- x %>% fn()


# 1 Date Time   01.01.2009 00:10:00 Date-time reference
# 2 p (mbar)    996.52  The pascal SI derived unit of pressure used to
#   quantify internal pressure. Meteorological reports typically state atmospheric
# pressure in millibars.
# 3 T (degC)    -8.02   Temperature in Celsius
# 4 Tpot (K)    265.4   Temperature in Kelvin
# 5 Tdew (degC) -8.9    Temperature in Celsius relative to humidity.
#   Dew Point is a measure of the absolute amount of water in the air, the DP is the
# temperature at which the air cannot hold all the moisture in it and water condenses.
# 6 rh (%)  93.3    Relative Humidity is a measure of how saturated the air is
#   with water vapor, the %RH determines the amount of water contained
#   within collection objects.
# 7 VPmax (mbar)    3.33    Saturation vapor pressure
# 8 VPact (mbar)    3.11    Vapor pressure
# 9 VPdef (mbar)    0.22    Vapor pressure deficit
# 10    sh (g/kg)   1.94    Specific humidity
# 11    H2OC (mmol/mol) 3.12    Water vapor concentration
# 12    rho (g/m ** 3)  1307.75 Airtight
# 13    wv (m/s)    1.03    Wind speed
# 14    max. wv (m/s)   1.75    Maximum wind speed
# 15    wd (deg)    152.3   Wind direction in degrees
```

```r
colnames(full_df)=c("dateTime", "pressure", "temperature",  "temp_Kelvin",  "dewPoint",
                    "relaHumidity", "svaporPress", "vaporPress", "vaporPressDef",
                    "specificHumid", "waterVapor", "airtight", "windSpeed",
                    "maxWindSpeed", "windDirection")


# plot the whole data set of the temperature time series
plot(temperature ~ dateTime, data = full_df, pch = 20, cex = .3)
```




```r
# Plotting the first 10 days of the temperature time series
plot(temperature ~ dateTime, data = full_df[1:1440, ])
```




```
### Use the  50% of the data for training, the following 25% for validation,
# and the last 25% for testing

num_train_samples <- round(nrow(full_df) * .5)
num_val_samples <- round(nrow(full_df) * 0.25)
num_test_samples <- nrow(full_df) - num_train_samples - num_val_samples

train_df <- full_df[seq(num_train_samples), ]
val_df <- full_df[seq(from = nrow(train_df) + 1, length.out = num_val_samples), ]
test_df <- full_df[seq(to = nrow(full_df), length.out = num_test_samples), ]

cat("num_train_samples:", nrow(train_df), "\n")
```

```
## num_train_samples: 210226
```

```
cat("num_val_samples:", nrow(val_df), "\n")
```

```
## num_val_samples: 105113
```

```
cat("num_test_samples:", nrow(test_df), "\n")
```

```
## num_test_samples: 105112
```

```
#####################################  standardize data
input_data_colnames <- names(full_df) %>% setdiff(c("dateTime", "temp_Kelvin"))

train_col_mean = sapply(train_df[input_data_colnames], mean)  # normalize the data
train_col_sd = sapply(train_df[input_data_colnames], sd)  # normalize the data

train_df_normalized = scale(train_df[input_data_colnames],
                            center = train_col_mean, scale = train_col_sd)
# normalize the training data

val_df_normalized = scale(val_df[input_data_colnames],
                          center = train_col_mean, scale = train_col_sd)
# normalize the validation data

test_df_normalized = scale(test_df[input_data_colnames],
                           center = train_col_mean, scale = train_col_sd)
# normalize the data testing data


dim(train_df_normalized)
```

```
## [1] 210226     13
```

```
dim(val_df_normalized)
```

```
## [1] 105113     13
```

```
dim(test_df_normalized)
```

```
## [1] 105112     13
```

```
#################################

ncol_input_data <- length(input_data_colnames)
```

```
library(tensorflow)
```

```
## Warning: package 'tensorflow' was built under R version 4.1.2
```

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 4.1.2
```

```
sampling_rate <- 6
sequence_length <- 120
delay <- sampling_rate * (sequence_length + 24 - 1)
batch_size <- 64      ### a batch of 256 samples, each containing 120 consecutive hours
# of input data, and targets is the corresponding array of 256 target temperatures
```

```
train_dataset <- timeseries_dataset_from_array(
              data = head(train_df_normalized, -delay), # truncate the last 'delay' records
                           targets = tail(train_df$temperature, -delay),       # remove the first
                           #'delay' records; so the input matches its target
              sampling_rate = sampling_rate,
              sequence_length = sequence_length,
              shuffle = TRUE,           #### the authors used shuffle=T,
                           #two consecutive sequences in a batch aren't necessarily temporally close
              batch_size = batch_size )
```

```
## Loaded Tensorflow version 2.10.0
```

```
val_dataset <- timeseries_dataset_from_array(
              data = head(as.matrix(val_df_normalized), -delay), # truncate the last
              #'delay' records
                           targets = tail(as.array(val_df$temperature), -delay),       # remove the
                           #first 'delay' records; so the input matches its target
              sampling_rate = sampling_rate,
              sequence_length = sequence_length,
              shuffle = TRUE,           #### the authors used shuffle=T,
                           #two consecutive sequences in a batch aren't necessarily temporally close
              batch_size = batch_size )


test_dataset <- timeseries_dataset_from_array(
              data = head(as.matrix(test_df_normalized), -delay), # truncate the
              #last 'delay' records
                           targets = tail(as.array(test_df$temperature), -delay),       # remove the
                           #first 'delay' records; so the input matches its target
              sampling_rate = sampling_rate,
              sequence_length = sequence_length,
              shuffle = TRUE,           #### the authors used shuffle=T, two consecutive
                           #sequences in a batch aren't necessarily temporally close
              batch_size = batch_size )
```

```
inputs <- layer_input(shape = c(sequence_length, ncol_input_data))

outputs <- inputs %>%
          #layer_lstm(32, recurrent_dropout = 0.5, return_sequences = TRUE) %>%
          layer_lstm(32, recurrent_dropout = 0.5) %>%
          layer_dense(1)

model <- keras_model(inputs, outputs)


callbacks_list <- list(
#callback_early_stopping(
#monitor = "val_loss", patience = 5),
callback_model_checkpoint(
filepath = "jena_lstm_dropout.keras2",
monitor = "val_mae", save_best_only = TRUE)
)
```

```
model <- keras_model(inputs, outputs)

model %>% compile(optimizer = "rmsprop",
                  loss = "mse",
                  metrics = "mae")


history2 <- model %>% fit(
  train_dataset,
  epochs = 100,
  validation_data = val_dataset,
  callbacks = callbacks_list
)
```

```
model2 <- load_model_tf("jena_lstm_dropout_64.keras2")
score <- model2 %>% evaluate(
  test_dataset,
  verbose = 1
)
cat('Test MSE:', score["loss"], '\n')
```

```
## Test MSE: 10.32954
```

```
cat('Test MAE:', score["mae"], '\n')
```

```
## Test MAE: 2.521762
```

Part (II) 30 Points

Run the stacked GRU model for Jena temperature forecasting.

- 1. Find out the MAE for the testing data.
- 2. Find out the MSE for the testing data.

```
inputs <- layer_input(shape = c(sequence_length, ncol_input_data))

outputs <- inputs %>%
       layer_gru(32, recurrent_dropout = 0.5, return_sequences = TRUE) %>%
       layer_gru(32, recurrent_dropout = 0.5) %>%
       layer_dropout(0.5) %>%
       layer_dense(1)


model <- keras_model(inputs, outputs)


callbacks_list2 <- list(
callback_early_stopping(
monitor = "val_mae", patience = 2),
```

```
callback_model_checkpoint(
filepath = "jena_stacked_gru_dropout.keras2",
monitor = "val_mae", save_best_only = TRUE)
)



model %>% compile(optimizer = "rmsprop",
     loss = "mse",
     metrics = "mae")

history <- model %>% fit(
                        train_dataset,
                        epochs = 50,
                        validation_data = val_dataset,
                        callbacks = callbacks_list2 )
```

```
model3 <- load_model_tf("jena_stacked_gru_dropout_64.keras2")
score <- model3 %>% evaluate(
  test_dataset,
  verbose = 1
)
cat('Test MSE:', score["loss"], '\n')
```

```
## Test MSE: 10.35234
```

```
cat('Test MAE:', score["mae"], '\n')
```

```
## Test MAE: 2.505722
```

```
# I also ran the same model with no stopping condition
model4 <- load_model_tf("jena_stacked_gru_dropout_noearlystopping.keras2")
score <- model4 %>% evaluate(
  test_dataset,
  verbose = 1
)
cat('Test MSE:', score["loss"], '\n')
```

```
## Test MSE: 9.847935
```

```
cat('Test MAE:', score["mae"], '\n')
```

```
## Test MAE: 2.486086
```

Part (III) 40 Points

Design your own model(different from the above example models, either GRU or LSTM) for Jena temperature forecasting, so that the MAE for the testing data is smaller than 2.62, which is the result by the naive method.

- 1. Find out the MAE for the testing data.
- 2. Find out the MSE for the testing data.

Lets try using the kerasTuneR package again to find an optimal set of parameters for a LSTM mode. I ran this concurrently on 2 separate computers, each generating/evaluating models and storing scores in a shared directory.

The input data was read in and processed in RConsole, and the following code chunk was saved as 'tuning_model2.R':

```
FLAGS <- flags(
  flag_numeric('dropout', 0.3),
  flag_integer('units1', 32),
  flag_integer('units2', 32),
  flag_integer('units3', 32),
  flag_numeric('rd1', 0.3),
  flag_numeric('rd2', 0.3),
  flag_numeric('rd3', 0.3)
)

build_model <- function() {
model <- keras_model_sequential()
  model %>% layer_gru(units = FLAGS$units1,
                      recurrent_dropout = FLAGS$rd1,
                      input_shape= c(sequence_length, ncol_input_data),
                      return_sequences = TRUE, activation='linear') %>%
           layer_gru(units =  FLAGS$units2,
                     recurrent_dropout = FLAGS$rd2,
           return_sequences = TRUE, activation='linear') %>%
           layer_gru(units =  FLAGS$units3,
                      recurrent_dropout = FLAGS$rd3,
          activation= 'linear') %>%
          layer_dropout(rate =FLAGS$dropout) %>%
          layer_dense(1, activation = 'linear')
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mean_squared_error",
    metrics = "mean_absolute_error"
  )
  model
}

model <- build_model()


early_stop <- callback_early_stopping(monitor = "val_mean_absolute_error",
                                      patience = 3)

# Fit the model and store training stats
history <- model %>% fit(
  train_dataset,
  epochs = 2,
  validation_data = val_dataset,
  verbose = 1,
```

```
  callbacks = list(early_stop)
)
plot(history)

score <- model %>% evaluate(
  test_dataset,
  verbose = 1
)

cat('Test loss:', score["loss"], '\n')
cat('Test accuracy:', score["mean_absolute_error"], '\n')
```

After saving this file, the following was ran on RConsole for both machines:

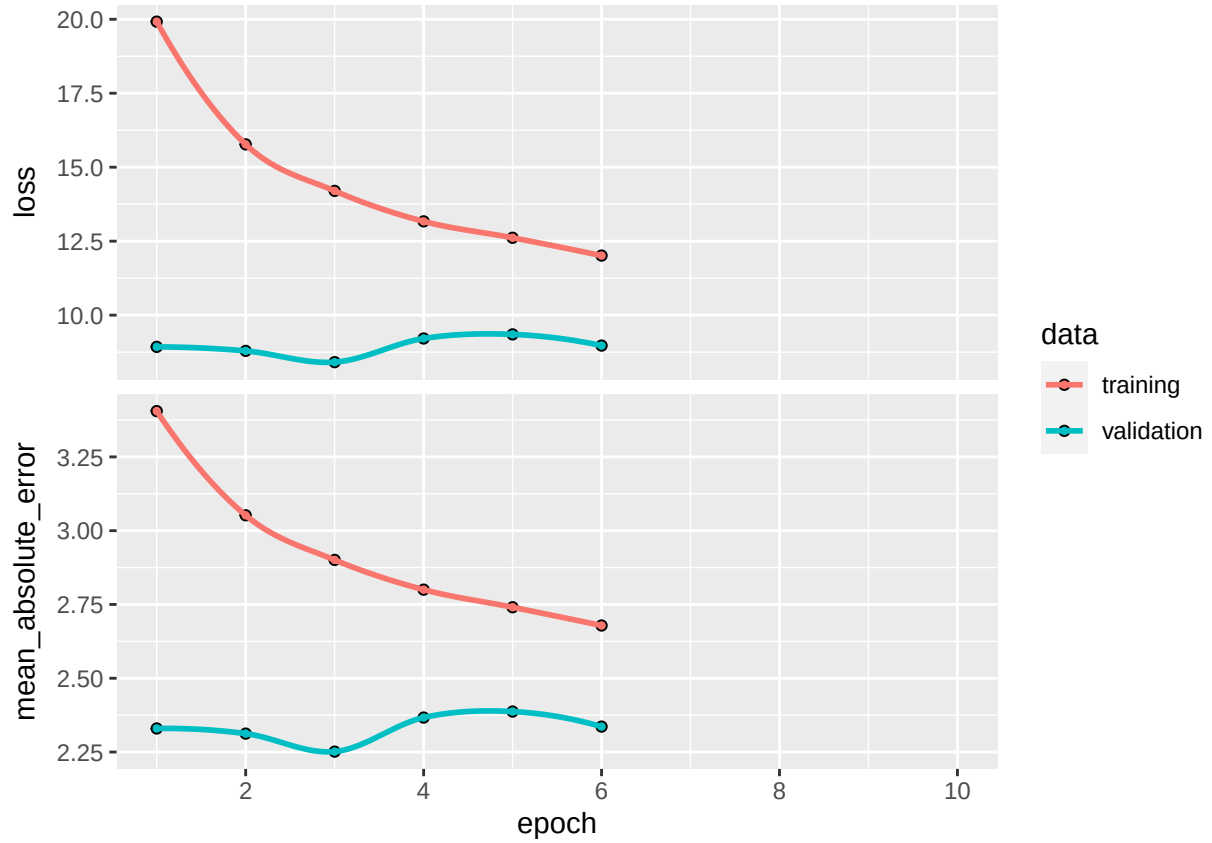
```
library(tfruns)
library(tfestimators)
par <- list(
  dropout = c(0.1,0.2,0.3,0.4,0.5),
  units1 = c(16,32,64,128,256),
  units2 = c(16,32,64,128,256),
  units3 = c(16,32,64,128,256),
  rd1 = c(0.1,0.2,0.3,0.4,0.5),
  rd2 = c(0.1,0.2,0.3,0.4,0.5),
  rd3 = c(0.1,0.2,0.3,0.4,0.5)
)
runs <- training_run("tuning_model.R", runs_dir = '_tuning', sample = 0.005, flags = par)
ls_runs(order = metric_val_mean_absolute_error, runs_dir = '_tuning')
```

This best model yielded a val_mae = 2.29 and a test_mae = 2.45; let's use these parameters and see if we can train the model to perform even better:

```
model <- keras_model_sequential()
  model %>% layer_gru(units = 16,
                      recurrent_dropout = .4,
                      input_shape= c(sequence_length, ncol_input_data),
                      return_sequences = TRUE, activation='linear') %>%
           layer_gru(units =  16,
                     recurrent_dropout = .3,
           return_sequences = TRUE, activation='linear') %>%
           layer_gru(units =  16,
                      recurrent_dropout = .2,
          activation= 'linear') %>%
          layer_dropout(rate =.5) %>%
          layer_dense(1, activation = 'linear')
  model %>% compile(
    optimizer = "rmsprop",
    loss = "mean_squared_error",
    metrics = "mean_absolute_error"
  )


early_stop <- callback_early_stopping(monitor = "val_mean_absolute_error",
                                      patience = 3)
```

```r
# Fit the model and store training stats
history <- model %>% fit(
  train_dataset,
  epochs = 10,
  validation_data = val_dataset,
  verbose = 1,
  callbacks = list(early_stop)
)
plot(history)
```

```
## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in sqrt(sum.squares/one.delta): NaNs produced

## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in sqrt(sum.squares/one.delta): NaNs produced

## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in sqrt(sum.squares/one.delta): NaNs produced

## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in simpleLoess(y, x, w, span, degree = degree, parametric =
## parametric, : Chernobyl! trL>n 6

## Warning in sqrt(sum.squares/one.delta): NaNs produced
```




```
score <- model %>% evaluate(
  test_dataset,
  verbose = 1
)

cat('Test loss:', score["loss"], '\n')
```

```
## Test loss: 10.12908
```

```
cat('Test accuracy:', score["mean_absolute_error"], '\n')
```

```
## Test accuracy: 2.511375
```

Not exactly the performance I hoped to gain. Playing with parameters and batch sizes did not yield quite the performance improvement that I hoped for! So refer to the first model found by keraTuneR for grading purposes.

I thoroughly enjoyed this class; it helped me build on the knowledge I previously gained with Dr. Cornwell and showed me a lot of new models and the mathematics behind them. I sincerly hope that I get to take another course with you (hopefully in the same subject area) before graduation.