Advanced_RNN:Weather Forecasting

SMANDUMU

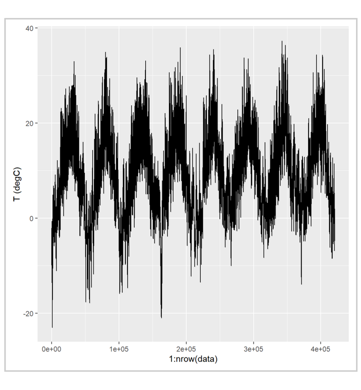
Improve weather forecasting for the problem

```
library(tibble)
library(readr)
library(keras)
data <- read_csv("jena_climate_2009_2016.csv")
glimpse(data)
```

```
## Rows: 420,551
## Columns: 15
## $ `Date Time`       <chr> "01.01.2009 00:10:00", "01.01.2009 00:20:00", "01...
## $ `p (mbar)`        <dbl> 996.52, 996.57, 996.53, 996.51, 996.51, 996.50, 9...
## $ `T (degC)`        <dbl> -8.02, -8.41, -8.51, -8.31, -8.27, -8.05, -7.62, ...
## $ `Tpot (K)`        <dbl> 265.40, 265.01, 264.91, 265.12, 265.15, 265.38, 2...
## $ `Tdew (degC)`     <dbl> -8.90, -9.28, -9.31, -9.07, -9.04, -8.78, -8.30, ...
## $ `rh (%)`          <dbl> 93.3, 93.4, 93.9, 94.2, 94.1, 94.4, 94.8, 94.4, 9...
## $ `VPmax (mbar)`    <dbl> 3.33, 3.23, 3.21, 3.26, 3.27, 3.33, 3.44, 3.44, 3...
## $ `VPact (mbar)`    <dbl> 3.11, 3.02, 3.01, 3.07, 3.08, 3.14, 3.26, 3.25, 3...
## $ `VPdef (mbar)`    <dbl> 0.22, 0.21, 0.20, 0.19, 0.19, 0.19, 0.18, 0.19, 0...
## $ `sh (g/kg)`       <dbl> 1.94, 1.89, 1.88, 1.92, 1.92, 1.96, 2.04, 2.03, 1...
## $ `H2OC (mmol/mol)` <dbl> 3.12, 3.03, 3.02, 3.08, 3.09, 3.15, 3.27, 3.26, 3...
## $ `rho (g/m**3)`    <dbl> 1307.75, 1309.80, 1310.24, 1309.19, 1309.00, 1307...
## $ `wv (m/s)`        <dbl> 1.03, 0.72, 0.19, 0.34, 0.32, 0.21, 0.18, 0.19, 0...
## $ `max. wv (m/s)`   <dbl> 1.75, 1.50, 0.63, 0.50, 0.63, 0.63, 0.63, 0.50, 0...
## $ `wd (deg)`        <dbl> 152.3, 136.1, 171.6, 198.0, 214.3, 192.7, 166.5, ...
```

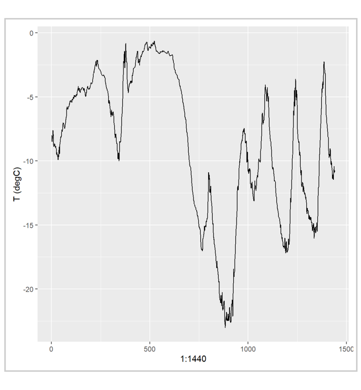
Here is the plot of temperature (in degrees Celsius) over time. On this plot, you can clearly see the yearly periodicity of temperature.

```
library(ggplot2)
ggplot(data, aes(x = 1:nrow(data), y = `T (degC)`)) + geom_line()
```



We clearly see the yearly periodicity of temperature.

```
ggplot(data[1:1440,], aes(x = 1:1440, y = `T (degC)`)) + geom_line()
```



We can see daily periodicity, especially evident for the last 4 days

Preparing the Data

We use the following parameter values:

- `lookback = 1440`, i.e. our observations will go back 10 days.
- `steps = 6`, i.e. our observations will be sampled at one data point per hour.
- `delay = 144`, i.e. our targets will be 24 hours in the future.

First, you'll convert the R data frame which we read earlier into a matrix of floating point values (we'll discard the first column which included a text timestamp):

```
data <- data.matrix(data[,-1])
```

We use the first 40000 timesteps as training data, so compute the mean and standard deviation for normalization only on this fraction of the data.

```
train_data <- data[1:40000,]
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
data <- scale(data, center = mean, scale = std)
```

The data generator you will use.Yields a list where `samples` is one batch of input data and `targets` is the corresponding array of target temperatures.

```
generator <- function(data, lookback, delay, min_index, max_index,
                      shuffle = FALSE, batch_size = 128, step = 6) {
  if (is.null(max_index))
    max_index <- nrow(data) - delay - 1
  i <- min_index + lookback
  function() {
    if (shuffle) {
      rows <- sample(c((min_index+lookback):max_index), size = batch_size)
    } else {
      if (i + batch_size >= max_index)
        i <<- min_index + lookback
      rows <- c(i:min(i+batch_size-1, max_index))
      i <<- i + length(rows)
    }

    samples <- array(0, dim = c(length(rows),
                                lookback / step,
                                dim(data)[[-1]]))
    targets <- array(0, dim = c(length(rows)))

    for (j in 1:length(rows)) {
      indices <- seq(rows[[j]] - lookback, rows[[j]] - 1,
                     length.out = dim(samples)[[2]])
      samples[j,,] <- data[indices,]
      targets[[j]] <- data[rows[[j]] + delay,2]
    }

    list(samples, targets)
  }
}
```

Now we use the abstract `generator` function to instantiate three generators: one for training, one for validation, and one for testing.

```
lookback <- 1440
step <- 6
delay <- 144
batch_size <- 128
train_gen <- generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 1,
  max_index = 20000,
  shuffle = TRUE,
  step = step,
  batch_size = batch_size
)
val_gen = generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 20001,
  max_index = 30000,
  step = step,
  batch_size = batch_size
)
test_gen <- generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 30001,
  max_index = 40000,
  step = step,
  batch_size = batch_size
)
# This is how many steps to draw from `val_gen`
# in order to see the whole validation set:
val_steps <- (22000 - 15001 - lookback) / batch_size
  # This is how many steps to draw from `test_gen`
# in order to see the whole test set:
test_steps <- (nrow(data) - 22001 - lookback) / batch_size
```

Non Machine Learning Baseline Model

```
evaluate_naive_method <- function() {
  batch_maes <- c()
  for (step in 1:val_steps) {
    c(samples, targets) %<-% val_gen()
    preds <- samples[,dim(samples)[[2]],2]
    mae <- mean(abs(preds - targets))
    batch_maes <- c(batch_maes, mae)
  }
  print(mean(batch_maes))
}
```

Dense Layered Model

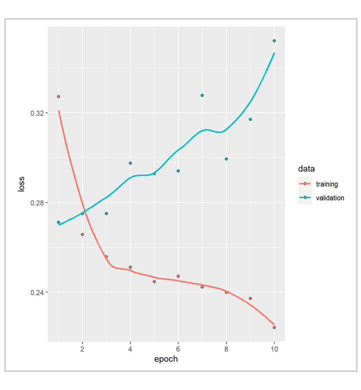
```
library(keras)

model <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(lookback / step, dim(data)[-1])) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)

history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 500,
  epochs = 10,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

Baseline Recurrent Model:gru

```
model <- keras_model_sequential() %>%
  layer_gru(units = 32, input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_dense(units = 1)
model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)
history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 50,
  epochs = 10,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

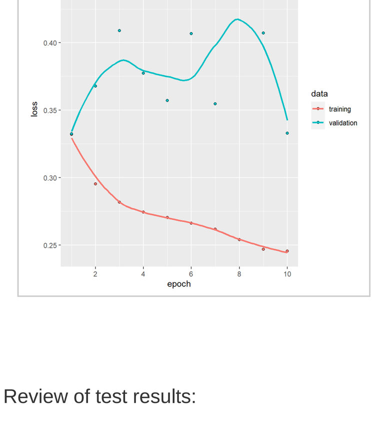


Improvements to Model : Recurrent Model with Dropout and Stacked gru Layers

```
model <- keras_model_sequential() %>%
  layer_gru(units = 64,
            dropout = 0.2,
            recurrent_dropout = 0.6,
            return_sequences = TRUE,
            input_shape = list(NULL, dim(data)[[-1]])) %>%
  layer_gru(units = 128,
            activation = "relu",
            dropout = 0.2,
            recurrent_dropout = 0.6) %>%
    layer_dense(units = 32, activation = "relu") %>%
  # Added dense layer 32 units
  layer_dense(units = 64, activation = "relu") %>%
  # Added dense layer 64 units
  layer_dense(units = 1)
model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)
history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 100,
  epochs = 10,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

```
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```

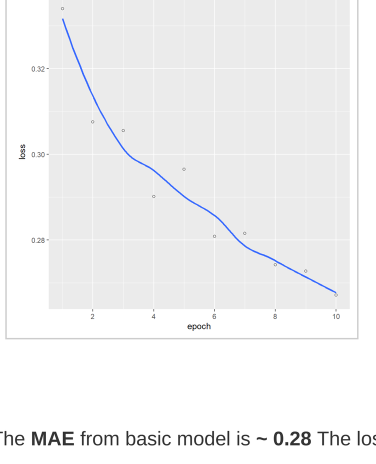


Review of test results:

```
history_test <- model %>% fit_generator(
  test_gen,
  steps_per_epoch = 100,
  epochs = 10,
   validation_steps = test_steps
)
```

```
plot(history_test)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



The **MAE** from basic model is **~ 0.28** The loss of test acheived **~ 26** at **Epoch 10** and we see that it has the minimum loss