TS

dl

22/4/2022

In this example, we'll be playing with a weather timeseries dataset recorded at the Weather Station at the Max-Planck-Institute for Biogeochemistry in Jena, Germany. In this dataset, 14 different quantities (such air temperature, atmospheric pressure, humidity, wind direction, and so on) were recorded every 10 minutes, over several years. The original data goes back to 2003, but this example is limited to data from 2009–2016.

This dataset is perfect for learning to work with numerical timeseries. We'll use it to build a model that takes as input some data from the recent past (a few days' worth of data points) and predicts the air temperature 24 hours in the future.

```
library(tibble)
library(readr)
library(ggplot2)
library(keras)
```

```
data_dir <- "~/deeplearn/unitat5/time_series"
fname <- file.path(data_dir, "jena_climate_2009_2016.csv")
data <- read_csv(fname)
```

```
## Rows: 420451 Columns: 15
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (1): Date Time
## dbl (14): p (mbar), T (degC), Tpot (K), Tdew (degC), rh (%), VPmax (mbar), V...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

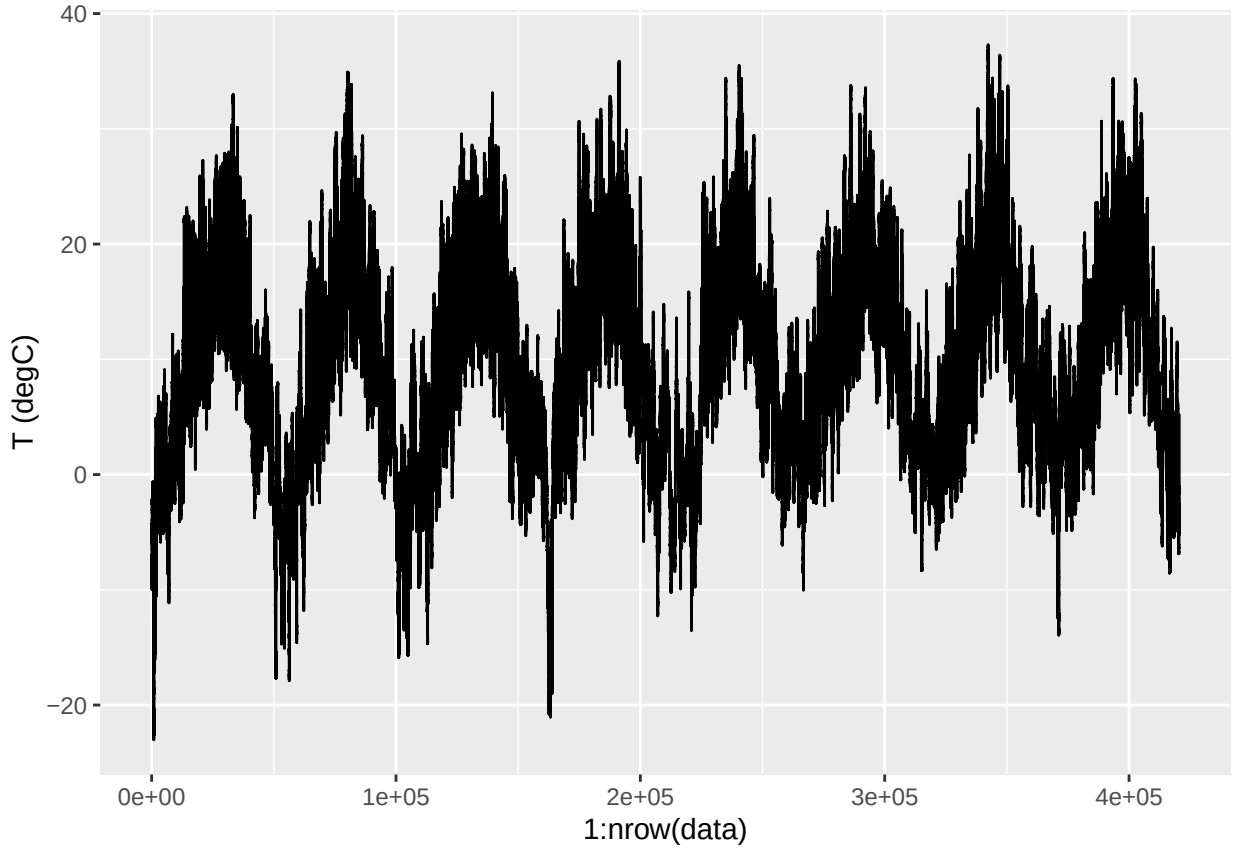
```
glimpse(data) # from tibble package
```

```
## Rows: 420,451
## Columns: 15
## $ `Date Time`       <chr> "01.01.2009 00:10:00", "01.01.2009 00:20:00", "01.01~
## $ `p (mbar)`        <dbl> 996.52, 996.57, 996.53, 996.51, 996.51, 996.50, 996.~
## $ `T (degC)`        <dbl> -8.02, -8.41, -8.51, -8.31, -8.27, -8.05, -7.62, -7.~
## $ `Tpot (K)`        <dbl> 265.40, 265.01, 264.91, 265.12, 265.15, 265.38, 265.~
## $ `Tdew (degC)`     <dbl> -8.90, -9.28, -9.31, -9.07, -9.04, -8.78, -8.30, -8.~
## $ `rh (%)`          <dbl> 93.3, 93.4, 93.9, 94.2, 94.1, 94.4, 94.8, 94.4, 93.8~
## $ `VPmax (mbar)`    <dbl> 3.33, 3.23, 3.21, 3.26, 3.27, 3.33, 3.44, 3.44, 3.36~
## $ `VPact (mbar)`    <dbl> 3.11, 3.02, 3.01, 3.07, 3.08, 3.14, 3.26, 3.25, 3.15~
## $ `VPdef (mbar)`    <dbl> 0.22, 0.21, 0.20, 0.19, 0.19, 0.19, 0.18, 0.19, 0.21~
```

```
## $ `sh (g/kg)`       <dbl> 1.94, 1.89, 1.88, 1.92, 1.92, 1.96, 2.04, 2.03, 1.97~
## $ `H2OC (mmol/mol)` <dbl> 3.12, 3.03, 3.02, 3.08, 3.09, 3.15, 3.27, 3.26, 3.16~
## $ `rho (g/m**3)`    <dbl> 1307.75, 1309.80, 1310.24, 1309.19, 1309.00, 1307.86~
## $ `wv (m/s)`        <dbl> 1.03, 0.72, 0.19, 0.34, 0.32, 0.21, 0.18, 0.19, 0.28~
## $ `max. wv (m/s)`   <dbl> 1.75, 1.50, 0.63, 0.50, 0.63, 0.63, 0.63, 0.50, 0.75~
## $ `wd (deg)`        <dbl> 152.3, 136.1, 171.6, 198.0, 214.3, 192.7, 166.5, 118~
```

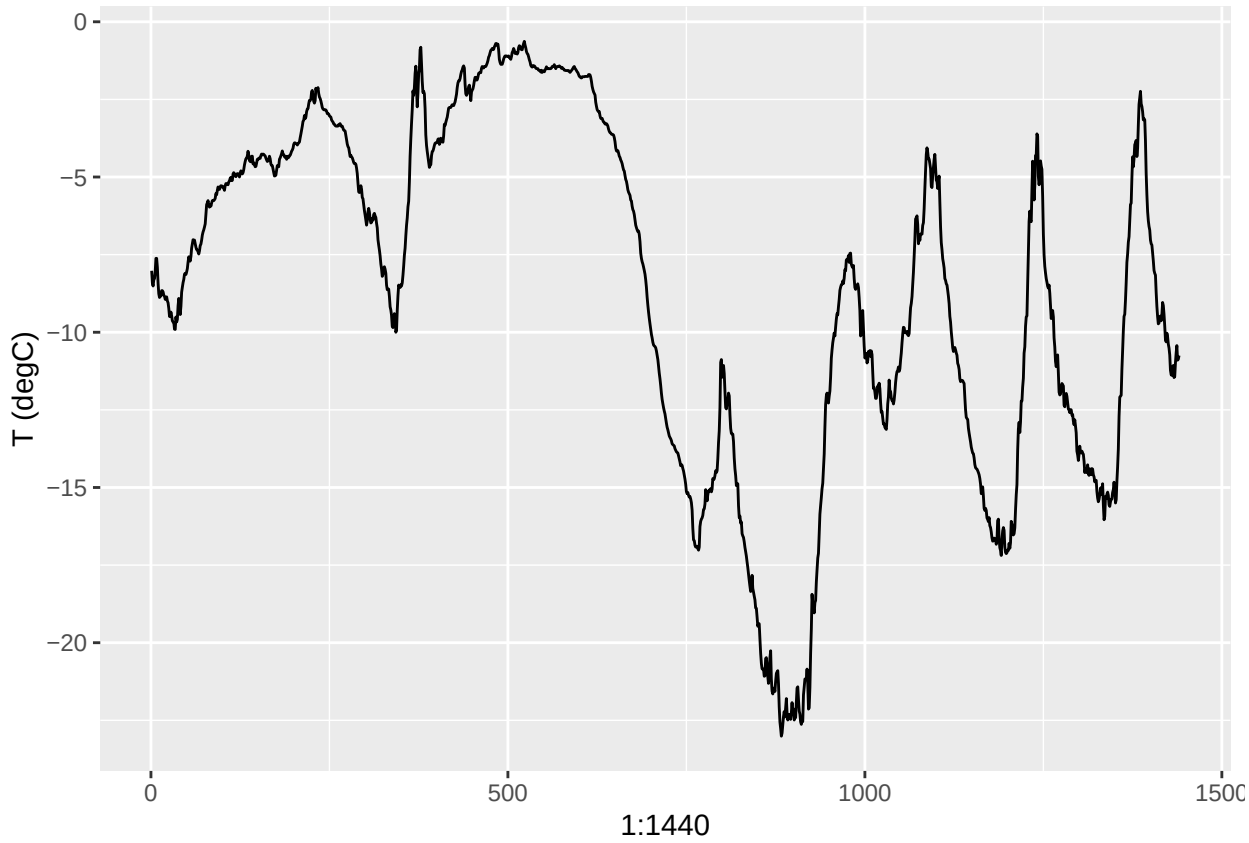
Here is the plot of temperature (in degrees Celsius) over time. On this plot, you can clearly see the yearly periodicity of temperature.

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



Here is a more narrow plot of the first 10 days of temperature data. Because the data is recorded every 10 minutes, you get 144 data points per day.

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



On this plot, you can see daily periodicity, especially evident for the last 4 days. Also note that this 10-day period must be coming from a fairly cold winter month.

If you were trying to predict average temperature for the next month given a few months of past data, the problem would be easy, due to the reliable year-scale periodicity of the data. But looking at the data over a scale of days, the temperature looks a lot more chaotic. Is this timeseries predictable at a daily scale? Let's find out.

Preparing the data

The exact formulation of the problem will be as follows: given data going as far back as lookback timesteps (a timestep is 10 minutes) and sampled every steps timesteps, can you predict the temperature in delay timesteps? You'll use the following parameter values:

-lookback = 720—Observations will go back 5 days. -steps = 6—Observations will be sampled at one data point per hour. -delay = 144—Targets will be 24 hours in the future.

To get started, you need to do two things:

- Preprocess the data to a format a neural network can ingest. This is easy: the data is already numerical, so you don't need to do any vectorization. But each timeseries in the data is on a different scale (for example, temperature is typically between -20 and +30, but pressure, measured in mbar, is around 1,000). You'll normalize each timeseries independently so that they all take small values on a similar scale.
- Write a `generator function` that takes the current array of float data and yields batches of data from the recent past, along with a target temperature in the future. Because the samples in the dataset are

highly redundant (sample N and sample $N+1$ will have most of their timesteps in common), it would be wasteful to explicitly allocate every sample. Instead, you'll generate the samples on the fly using the original data.

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'll then preprocess the data by subtracting the mean of each timeseries and dividing by the standard deviation. We're going to use the first 200,000 timesteps as training data, so compute the mean and standard deviation for normalization only on this fraction of the data.

```
train_data <- data[1:200000,]
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
data <- scale(data, center = mean, scale = std)
```

The data generator we'll use. It yields a list (samples, targets), where samples is one batch of input data and targets is the corresponding array of target temperatures. It takes the following arguments:

- data—The original array of floating-point data, which we normalized.
- lookback—How many timesteps back the input data should go.
- delay—How many timesteps in the future the target should be.
- min_index and max_index—Indices in the data array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another for testing.
- shuffle —Whether to shuffle the samples or draw them in chronological order.
- batch_size —The number of samples per batch.
- step —The period, in timesteps, at which you sample data. You'll set it 6 in order to draw one data point every hour.

```
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, 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]],
length.out = dim(samples)[[2]])
samples[j,,] <- data[indices,]
targets[[j]] <- data[rows[[j]] + delay,2]
}

list(samples, targets)   # output
}
}
```

The i variable contains the state that tracks next window of data to return, so it is updated using superassignment (e.g. i $<$ i + length(rows)).

Now, let's use the abstract generator function to instantiate three generators: one for training, one for validation, and one for testing. Each will look at different temporal segments of the original data: the training generator looks at the first 200,000 timesteps, the validation generator looks at the following 100,000, and the test generator looks at the remainder.

```
lookback <- 1440
step <- 6
delay <- 144
batch_size <- 128

train_gen <- generator(
data,
lookback = lookback,
delay = delay,
min_index = 1,
max_index = 200000,
shuffle = TRUE,
step = step,
batch_size = batch_size
)

val_gen = generator(
data,
lookback = lookback,
delay = delay,
min_index = 200001,
max_index = 300000,
step = step,
batch_size = batch_size
)

test_gen <- generator(
data,
lookback = lookback,
delay = delay,
min_index = 300001,
max_index = NULL,
step = step,
batch_size = batch_size
```

```
)

val_steps <- (300000 - 200001 - lookback) / batch_size
test_steps <- (nrow(data) - 300001 - lookback) / batch_size
```

A common-sense, non-machine-learning baseline.

Before you start using black-box deep-learning models to solve the temperature-prediction problem, let's try a simple, common-sense approach. It will serve as a sanity check, and it will establish a baseline that you'll have to beat in order to demonstrate the usefulness of more advanced machine-learning models. Such common-sense baselines can be useful when you're approaching a new problem for which there is no known solution (yet).

In this case, the temperature timeseries can safely be assumed to be continuous (the temperatures tomorrow are likely to be close to the temperatures today) as well as periodical with a daily period. Thus a common-sense approach is to always predict that the temperature 24 hours from now will be equal to the temperature right now. Let's evaluate this approach, using the mean absolute error (MAE) metric:

```
evaluate_naive_method <- function() {
batch_maes <- c()

for (step in 1:val_steps) {
c(samples, targets) %<-% val_gen()
preds <- samples[,dim(samples)[[2]],2]  # T dia
mae <- mean(abs(preds - targets)) #  targets = T future
batch_maes <- c(batch_maes, mae)
}


print(mean(batch_maes))
}

evaluate_naive_method()
```

```
## [1] 0.2775185
```

This yields an MAE of 0.29. Because the temperature data has been normalized to be centered on 0 and have a standard deviation of 1, this number isn't immediately interpretable. It translates to an average absolute error of 0.29 * temperature_std degrees Celsius: 2.57C.

```
celsius_mae <- 0.29 * std[[2]]
celsius_mae
```

```
## [1] 2.567231
```

Basic approach

In the same way that it's useful to establish a common-sense baseline before trying machine-learning approaches, it's useful to try simple, cheap machine-learning models (such as small, densely connected networks)

before looking into complicated and computationally expensive models such as RNNs. This is the best way to make sure any further complexity you throw at the problem is legitimate and delivers real benefits. The following listing shows a fully connected model that starts by flattening the data and then runs it through two dense layers. Note the lack of activation function on the last dense layer, which is typical for a regression problem. You use MAE as the loss. Because you're evaluating on the exact same data and with the exact same metric you did with the common-sense approach, the results will be directly comparable.

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

```
## Loaded Tensorflow version 2.4.1
```

```
summary(model)
```

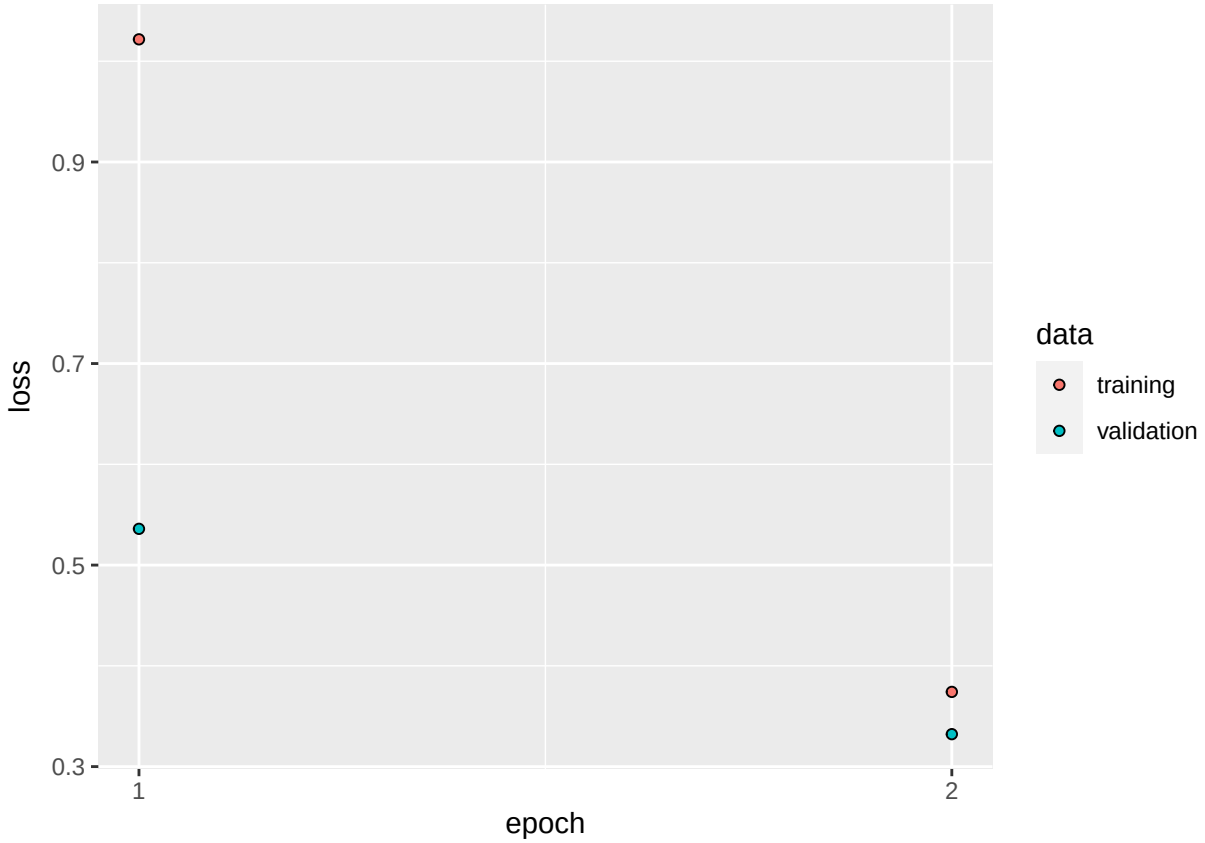
```
## Model: "sequential"
## _________________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =================================================================================
## flatten (Flatten)               (None, 3360)                  0
## _________________________________________________________________________________
## dense_1 (Dense)                 (None, 32)                    107552
## _________________________________________________________________________________
## dense (Dense)                   (None, 1)                     33
## =================================================================================
## Total params: 107,585
## Trainable params: 107,585
## Non-trainable params: 0
## _________________________________________________________________________________
```

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

```
history <- model %>% fit_generator(
train_gen,
steps_per_epoch = 500,
epochs = 2, #20
validation_data = val_gen,
validation_steps = val_steps
)
```

```
## Warning in fit_generator(., train_gen, steps_per_epoch = 500, epochs = 2, :
## `fit_generator` is deprecated. Use `fit` instead, it now accept generators.
```

```
plot(history)
```



Some of the validation losses are close to the no-learning baseline, but not reliably. This goes to show the merit of having this baseline in the first place: it turns out to be not easy to outperform. Your common sense contains a lot of valuable information that a machine-learning model doesn't have access to.

A first recurrent baseline

```
model <- keras_model_sequential() %>%
layer_gru(units = 32, dropout = 0.2, recurrent_dropout = 0.2,
input_shape = list(NULL, dim(data)[[-1]])) %>%
layer_dense(units = 1)
```

```
summary(model)
```

```
## Model: "sequential_1"
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## gru (GRU)                    (None, 32)                4608
## _________________________________________________________________________
## dense_2 (Dense)              (None, 1)                 33
## =========================================================================
## Total params: 4,641
## Trainable params: 4,641
```

```
## Non-trainable params: 0
## _________________________________________________________________________
```

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

```
history <- model %>% fit(
train_gen,
steps_per_epoch = 500,
epochs = 2, #20
validation_data = val_gen,
validation_steps = val_steps
)
```

The new validation MAE of ~0.265 (before you start significantly overfitting) translates to a mean absolute error of 2.35C after denormalization. That's a solid gain on the initial error of 2.57C, but you probably still have a bit of a margin for improvement.

Using recurrent dropout to fight overfitting

It's evident from the training and validation curves that the model is overfitting: the training and validation losses start to diverge considerably after a few epochs. You're already familiar with a classic technique for fighting this phenomenon: dropout, which randomly zeros out input units of a layer in order to break happenstance correlations in the training data that the layer is exposed to.

```
model <- keras_model_sequential() %>%
layer_gru(units = 32, dropout = 0.2, recurrent_dropout = 0.2,
input_shape = list(NULL, dim(data)[[-1]])) %>%
layer_dense(units = 1)
```

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

```
history <- model %>% fit(
train_gen,
steps_per_epoch = 500,
epochs = 2, #40
validation_data = val_gen,
validation_steps = val_steps
)
```