Tutorial: Deep Learning

by Kevin Kotzé

This tutorial will make use of the files in the `T18-deep.zip` folder. Thereafter, you will need to open the `T18-deep.Rproj` file to open the project, before opening the `tut18-temp.R` script.

1 Forecasting temperature with recurrent neural networks

This example is largely taken from the book _Deep Learning with R _by François Chollet & J. J. Allaire. The weather time series dataset recorded at the Weather Station at the Max-Planck-Institute for Biogeochemistry in Jena, Germany: http://www.bgc-jena.mpg.de/wetter/ (http://www.bgc-jena.mpg.de/wetter/).

In this dataset, fourteen different quantities (such air temperature, atmospheric pressure, humidity, wind direction, etc.) are recorded every ten minutes, over several years. The original data goes back to 2003, but we limit ourselves to data from 2009-2016. This dataset is perfect for learning to work with numerical time series. We will 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.

```
rm(list = ls())
graphics.off()
```

Thereafter, we will need to make use of the `keras` package, so we make use of the `library` command.

```
library(keras)
```

If you need install this package run the following routine: `install.packages("keras")`. It is usually a good idea to install this package from a new **RStudio** session, so you may want to close and restart **RStudio** before completing this installation procedure. This will also install an interface with Python, with the aid of the `reticulate` package as well as the backend engine, *TensorFlow*. If you have already installed **Anaconda** then you should not want to elect the option to install **MiniConda**.

After this is complete you then need to run an additional installation command. To complete the default CPU-based installation of *Keras* and *TensorFlow*, we would use the command:

```
install_keras()
```

```
##
## Installation complete.
```

However, if your computer has dedicated graphics card then you would use the command: `install_keras(tensorflow = "gpu")`. Once this is complete, then you are able to load the additional libraries.

```
library(tidyverse)
library(lubridate)
```

As this data is contained in a `.csv` file we need to use the `read_csv()` command.

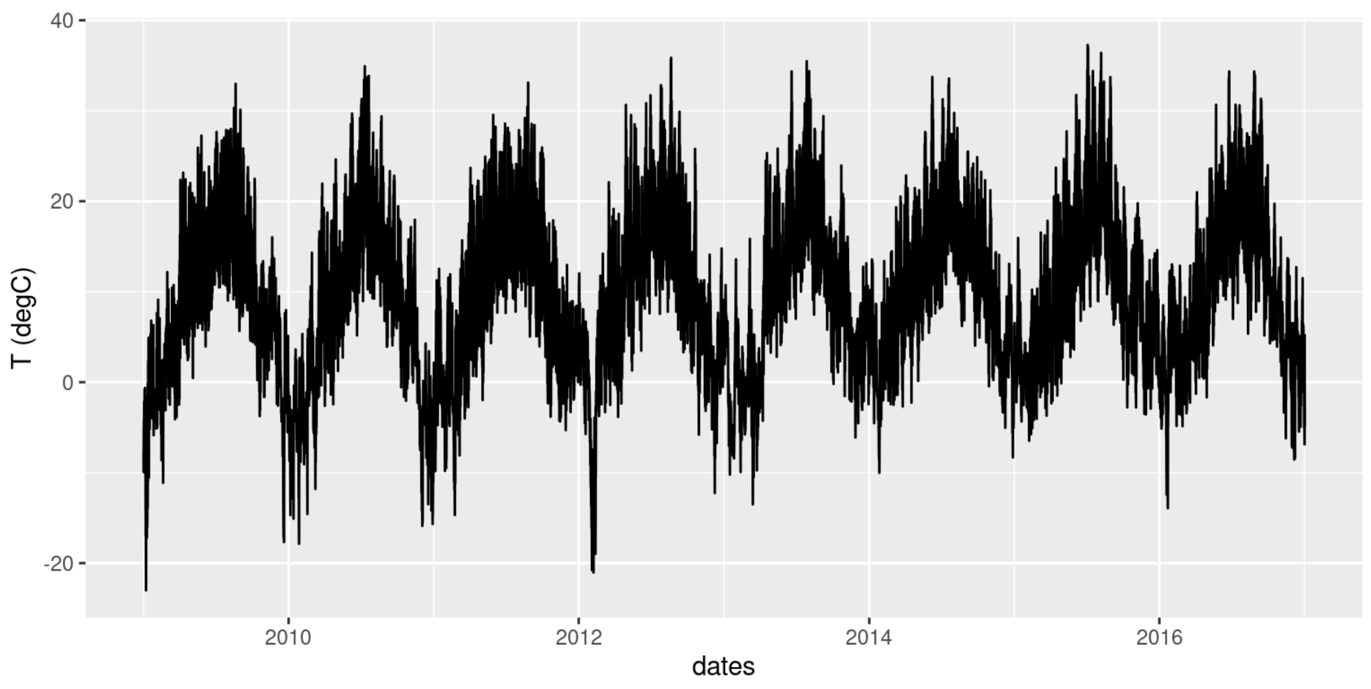
```
data <- read_csv("jena_climate_2009_2016.csv")
```

To convert the dates into a series of dates that the computer will recognise:

```
data <- data %>%
  mutate(dates = dmy_hms(str_replace_all(data$`Date Time`, "[.]", "-")))
```

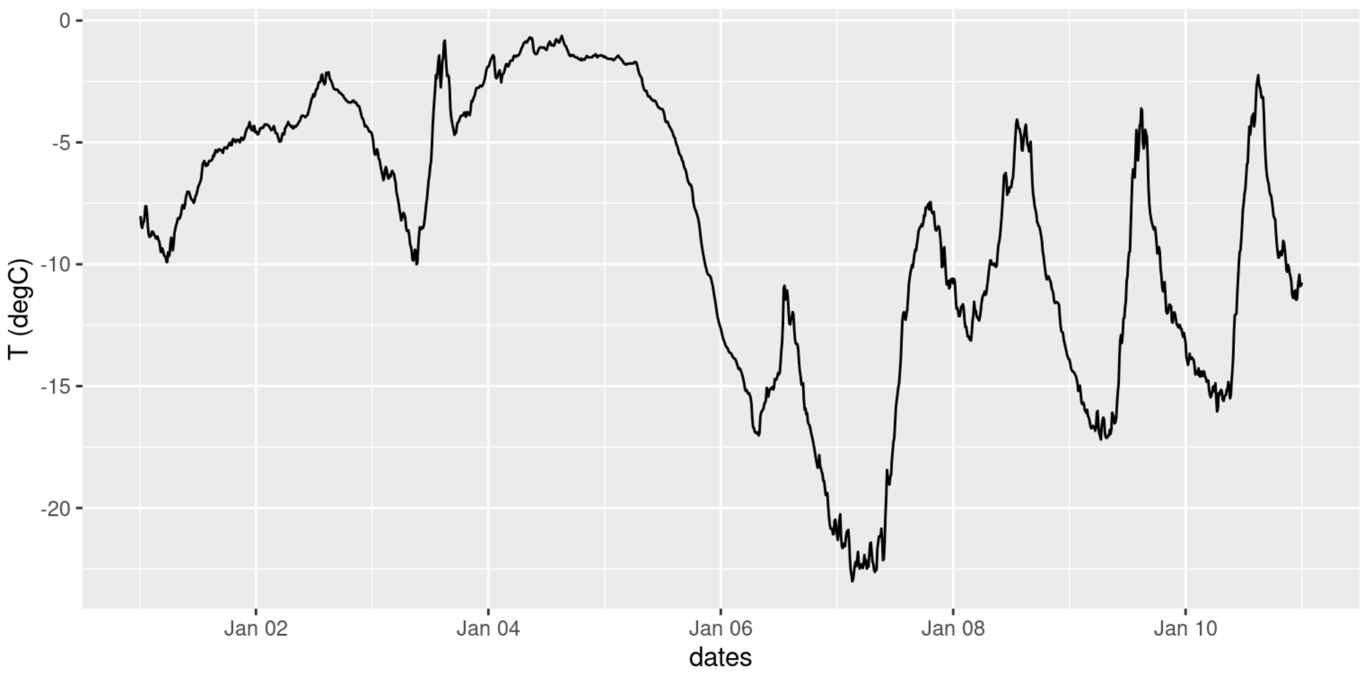
To ensure that this has all been completed correctly, we can plot the data. This shows the temperature (in degrees Celsius) over time. On this plot, you can clearly see the yearly periodicity of temperature.

```
ggplot(data, aes(x = dates, y = `T (degC)`)) + geom_line()
```



Or for a smaller subset of the data, where we plot the first ten days of temperature data (since the data is recorded every ten minutes, we get 144 data points per day):

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



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

Looking at this data sample the temperature looks relatively chaotic. To see if we are able to forecast the temperature we use a RNN model in the following example

1.1 Preparing the data

To construct this forecasting exercise we use the following setup:

- `lookback = 1440`, i.e. our observations will go back 10 days (in-sample period)
- `steps = 6`, i.e. our observations will be sampled at one data point per hour (aggregate data from 10 mins to 1 hour)
- `delay = 144`, i.e. our targets will be 24 hours in the future (want to generate a forecast 24 hours ahead)

To get started, we need to do two things:

- Preprocess the data into a format the neural network can ingest. This is easy: the data is already numerical, so we don't need to do any vectorization. However each time series in the data is on a different scale (e.g. temperature is typically between -20 and +30, but pressure, measured in mbar, is around 1000). So we will normalize each time series 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. Since the samples in the dataset are highly redundant (sample *N* and sample *N + 1* will have most of their time-steps in common), it would be wasteful to explicitly allocate every sample. Instead, we'll generate the samples on the fly using the original data.

First, we'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):

```r
data <- data %>%
  dplyr::select(-dates)
data <- data.matrix(data[,-1])
```

We'll then preprocess the data by subtracting the mean of each time series and dividing by the standard deviation. This is a standard practice in many machine learning applications. we're going to use the first 200,000 time-steps as training data, so compute the mean and standard deviation for normalization only on this fraction of the data.

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

Now here is 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 that we normalized
- `lookback` – How many time-steps back the input data should go
- `delay` – How many time-steps in the future the target should be
- `min_index` and `max_index` – Indices in the `data` array that delimit which time-steps to draw from (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 time-steps, at which we sample data which is set at 6 in order to draw one data point every hour

```r
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)
  }
}
```

The `i` variable contains the state that tracks the 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 create three data 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 time-steps, 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
)

# This is how many steps to draw from `val_gen`
# in order to see the whole validation set:
val_steps <- (300000 - 200001 - 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) - 300001 - lookback) / batch_size
```

1.2 A common sense, non-machine learning baseline

The baseline model that is used for comparative purposes is a random walk, where it is assumed that the temperature 24 hours from now will be equal to the current temperature. Let's evaluate this approach, using the Mean Absolute Error metric (MAE), which is simply equal to:

```
mean(abs(preds - targets))
```

So we can create an evaluation loop:

```r
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))
}
```

It yields a MAE of 0.29. Since our temperature data has been normalized to be centered on 0 and has a standard deviation of one, this number is not immediately interpretable. It translates to an average absolute error of `0.29 * temperature_std` degrees Celsius, i.e. 2.57˚C. That's a fairly large average absolute error – now the game is to leverage our knowledge of deep learning to do better.

1.3 A basic machine learning 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. Since we're evaluating on the exact same data and with the exact same metric as the baseline random-walk approach, the results will be directly comparable.

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

# Compile the model (specify optimistion and loss function)
model %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "mae"
)
```

To view a summary of the model:

```r
summary(model)
```

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

Take note of the total number of parameters.

```
# Train the model
history <- model %>% fit_generator(
  train_gen,
  steps_per_epoch = 100,
  epochs = 10,
  validation_data = val_gen,
  validation_steps = val_steps
)
```

```
Epoch 1/10

        1/100    [..............................]    -    ETA:    14s    -    loss:    0.884
1⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        2/100    [..............................]    -    ETA:    19s    -    loss:    0.959
9⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        3/100    [..............................]    -    ETA:    19s    -    loss:    0.890
1⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        4/100    [>.............................]    -    ETA:    18s    -    loss:    0.887
9⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        5/100    [>.............................]    -    ETA:    19s    -    loss:    0.923
0⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        6/100    [>.............................]    -    ETA:    19s    -    loss:    0.966
0⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        7/100    [=>............................]    -    ETA:    18s    -    loss:    0.956
7⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        8/100    [=>............................]    -    ETA:    18s    -    loss:    0.960
6⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
        9/100    [=>............................]    -    ETA:    17s    -    loss:    0.936
4⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     10/100    [==>...........................]    -    ETA:    17s    -    loss:    0.931
4⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     11/100    [==>...........................]    -    ETA:    17s    -    loss:    0.937
1⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     12/100    [==>...........................]    -    ETA:    17s    -    loss:    0.962
4⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     13/100    [==>...........................]    -    ETA:    16s    -    loss:    0.977
2⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     14/100    [===>..........................]    -    ETA:    16s    -    loss:    0.968
3⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     15/100    [===>..........................]    -    ETA:    15s    -    loss:    0.985
7⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     16/100    [===>..........................]    -    ETA:    15s    -    loss:    1.002
8⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     17/100    [====>.........................]    -    ETA:    15s    -    loss:    0.985
6⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     18/100    [====>.........................]    -    ETA:    15s    -    loss:    0.979
4⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     19/100    [====>.........................]    -    ETA:    14s    -    loss:    0.970
1⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     20/100    [=====>........................]    -    ETA:    14s    -    loss:    0.963
3⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     21/100    [=====>........................]    -    ETA:    14s    -    loss:    0.969
3⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     22/100    [=====>........................]    -    ETA:    14s    -    loss:    0.983
5⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     23/100    [=====>........................]    -    ETA:    13s    -    loss:    0.991
4⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     24/100    [======>.......................]    -    ETA:    13s    -    loss:    0.996
4⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     25/100    [======>.......................]    -    ETA:    13s    -    loss:    0.993
1⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     26/100    [======>.......................]    -    ETA:    13s    -    loss:    0.986
0⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰⍰
     27/100    [=======>......................]    -    ETA:    13s    -    loss:    0.981
```

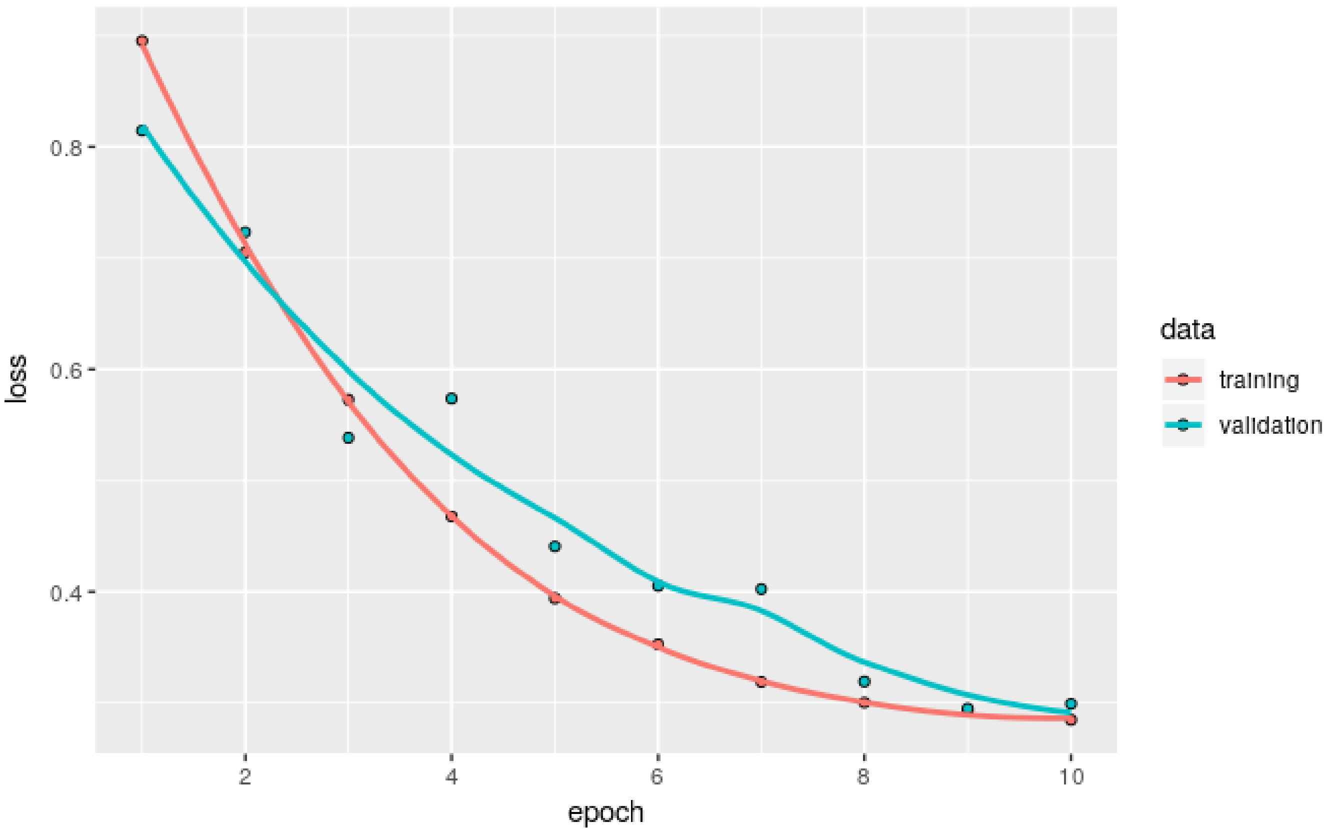
```
2���������������������������������������������������������������������
     28/100     [=======>......................]     -     ETA:     13s     -     loss:     0.981
5���������������������������������������������������������������������
     29/100     [=======>......................]     -     ETA:     12s     -     loss:     0.989
5���������������������������������������������������������������������
     30/100     [========>.....................]     -     ETA:     12s     -     loss:     0.982
4���������������������������������������������������������������������
     31/100     [========>.....................]     -     ETA:     12s     -     loss:     0.975
1���������������������������������������������������������������������
     32/100     [========>.....................]     -     ETA:     12s     -     loss:     0.963
6���������������������������������������������������������������������
     33/100     [========>.....................]     -     ETA:     12s     -     loss:     0.958
6���������������������������������������������������������������������
     34/100     [=========>....................]     -     ETA:     11s     -     loss:     0.957
9���������������������������������������������������������������������
     35/100     [=========>....................]     -     ETA:     11s     -     loss:     0.956
3���������������������������������������������������������������������
     36/100     [=========>....................]     -     ETA:     11s     -     loss:     0.960
6���������������������������������������������������������������������
     37/100     [==========>...................]     -     ETA:     11s     -     loss:     0.956
9���������������������������������������������������������������������
     38/100     [==========>...................]     -     ETA:     11s     -     loss:     0.955
5���������������������������������������������������������������������
     39/100     [==========>...................]     -     ETA:     10s     -     loss:     0.955
4���������������������������������������������������������������������
     40/100     [===========>..................]     -     ETA:     10s     -     loss:     0.957
4���������������������������������������������������������������������
     41/100     [===========>..................]     -     ETA:     10s     -     loss:     0.956
4���������������������������������������������������������������������
     42/100     [===========>..................]     -     ETA:     10s     -     loss:     0.955
6���������������������������������������������������������������������
     43/100     [===========>..................]     -     ETA:     10s     -     loss:     0.955
2���������������������������������������������������������������������
     44/100     [============>.................]     -     ETA:     9s     -     loss:     0.9547
�������������������������������������������������������������������
     45/100     [============>.................]     -     ETA:     9s     -     loss:     0.952
1��������������������������������������������������������������������
     46/100     [============>.................]     -     ETA:     9s     -     loss:     0.948
2��������������������������������������������������������������������
     47/100     [=============>................]     -     ETA:     9s     -     loss:     0.943
7��������������������������������������������������������������������
     48/100     [=============>................]     -     ETA:     9s     -     loss:     0.942
7��������������������������������������������������������������������
     49/100     [=============>................]     -     ETA:     9s     -     loss:     0.941
1��������������������������������������������������������������������
     50/100     [==============>...............]     -     ETA:     8s     -     loss:     0.937
8��������������������������������������������������������������������
     51/100     [==============>...............]     -     ETA:     8s     -     loss:     0.935
0��������������������������������������������������������������������
     52/100     [==============>...............]     -     ETA:     8s     -     loss:     0.937
3��������������������������������������������������������������������
     53/100     [==============>...............]     -     ETA:     8s     -     loss:     0.938
2��������������������������������������������������������������������
     54/100     [===============>..............]     -     ETA:     8s     -     loss:     0.940
5��������������������������������������������������������������������
     55/100     [===============>..............]     -     ETA:     7s     -     loss:     0.937
```

```
0�������������������������������������������������������������������������
     56/100  [===============>..............]  -  ETA:  7s  -  loss:  0.935
0�������������������������������������������������������������������������
     57/100  [================>.............]  -  ETA:  7s  -  loss:  0.933
1�������������������������������������������������������������������������
     58/100  [================>.............]  -  ETA:  7s  -  loss:  0.935
7�������������������������������������������������������������������������
     59/100  [================>.............]  -  ETA:  7s  -  loss:  0.934
6�������������������������������������������������������������������������
     60/100  [=================>............]  -  ETA:  7s  -  loss:  0.931
1�������������������������������������������������������������������������
     61/100  [=================>............]  -  ETA:  6s  -  loss:  0.929
8�������������������������������������������������������������������������
     62/100  [=================>............]  -  ETA:  6s  -  loss:  0.929
6�������������������������������������������������������������������������
     63/100  [=================>............]  -  ETA:  6s  -  loss:  0.926
7�������������������������������������������������������������������������
     64/100  [==================>...........]  -  ETA:  6s  -  loss:  0.927
2�������������������������������������������������������������������������
     65/100  [==================>...........]  -  ETA:  6s  -  loss:  0.926
4�������������������������������������������������������������������������
     66/100  [==================>...........]  -  ETA:  6s  -  loss:  0.926
5�������������������������������������������������������������������������
     67/100  [===================>..........]  -  ETA:  5s  -  loss:  0.925
4�������������������������������������������������������������������������
     68/100  [===================>..........]  -  ETA:  5s  -  loss:  0.924
5�������������������������������������������������������������������������
     69/100  [===================>..........]  -  ETA:  5s  -  loss:  0.924
7�������������������������������������������������������������������������
     70/100  [====================>.........]  -  ETA:  5s  -  loss:  0.924
6�������������������������������������������������������������������������
     71/100  [====================>.........]  -  ETA:  5s  -  loss:  0.919
9�������������������������������������������������������������������������
     72/100  [====================>.........]  -  ETA:  4s  -  loss:  0.916
5�������������������������������������������������������������������������
     73/100  [====================>.........]  -  ETA:  4s  -  loss:  0.915
7�������������������������������������������������������������������������
     74/100  [=====================>........]  -  ETA:  4s  -  loss:  0.917
0�������������������������������������������������������������������������
     75/100  [=====================>........]  -  ETA:  4s  -  loss:  0.917
6�������������������������������������������������������������������������
     76/100  [=====================>........]  -  ETA:  4s  -  loss:  0.917
2�������������������������������������������������������������������������
     77/100  [======================>.......]  -  ETA:  4s  -  loss:  0.914
8�������������������������������������������������������������������������
     78/100  [======================>.......]  -  ETA:  3s  -  loss:  0.913
0�������������������������������������������������������������������������
     79/100  [======================>.......]  -  ETA:  3s  -  loss:  0.912
4�������������������������������������������������������������������������
     80/100  [=======================>......]  -  ETA:  3s  -  loss:  0.913
0�������������������������������������������������������������������������
     81/100  [=======================>......]  -  ETA:  3s  -  loss:  0.911
6�������������������������������������������������������������������������
     82/100  [=======================>......]  -  ETA:  3s  -  loss:  0.907
6�������������������������������������������������������������������������
     83/100  [=======================>......]  -  ETA:  2s  -  loss:  0.904
```

```
7����������������������������������������������������������������������������
     84/100     [========================>.....]     -     ETA:     2s     -     loss:     0.905
2����������������������������������������������������������������������������
     85/100     [========================>.....]     -     ETA:     2s     -     loss:     0.906
3����������������������������������������������������������������������������
     86/100     [========================>.....]     -     ETA:     2s     -     loss:     0.908
1����������������������������������������������������������������������������
     87/100     [=========================>....]     -     ETA:     2s     -     loss:     0.907
9����������������������������������������������������������������������������
     88/100     [=========================>....]     -     ETA:     2s     -     loss:     0.908
4����������������������������������������������������������������������������
     89/100     [=========================>....]     -     ETA:     1s     -     loss:     0.905
4����������������������������������������������������������������������������
     90/100     [==========================>...]     -     ETA:     1s     -     loss:     0.903
9����������������������������������������������������������������������������
     91/100     [==========================>...]     -     ETA:     1s     -     loss:     0.902
5����������������������������������������������������������������������������
     92/100     [==========================>...]     -     ETA:     1s     -     loss:     0.903
5����������������������������������������������������������������������������
     93/100     [==========================>...]     -     ETA:     1s     -     loss:     0.901
5����������������������������������������������������������������������������
     94/100     [===========================>..]     -     ETA:     1s     -     loss:     0.900
0����������������������������������������������������������������������������
     95/100     [===========================>..]     -     ETA:     0s     -     loss:     0.897
9����������������������������������������������������������������������������
     96/100     [===========================>..]     -     ETA:     0s     -     loss:     0.897
5����������������������������������������������������������������������������
     97/100     [============================>.]     -     ETA:     0s     -     loss:     0.895
9����������������������������������������������������������������������������
     98/100     [============================>.]     -     ETA:     0s     -     loss:     0.894
8����������������������������������������������������������������������������
 99/100 [============================>.] - ETA: 0s - loss: 0.8956
```

Now let’s display the loss curves for validation and training:

```
plot(history)
```



Some of our validation losses get close to the random-walk baseline, but not very reliably.

You may ask, if there exists a simple, well-performing model to go from the data to the targets (our common sense baseline), why doesn’t the model we are training find it and improve on it? Simply put: because this simple solution is not what our training setup is looking for. The space of models in which we are searching for a solution, i.e. our hypothesis space, is the space of all possible 2-layer networks with the configuration that we defined. These networks are already fairly complicated. When looking for a solution with a space of complicated models, the simple well-performing baseline might be unlearnable, even if it’s technically part of the hypothesis space. That is a pretty significant limitation of machine learning in general: unless the learning algorithm is hard-coded to look for a specific kind of simple model, parameter learning can sometimes fail to find a simple solution to a simple problem.

A first recurrent baseline

Our first fully-connected approach didn’t do so well, but that doesn’t mean machine learning is not applicable to our problem. The approach above consisted in first flattening the timeseries, which removed the notion of time from the input data. Let us instead look at our data as what it is: a sequence, where causality and order matter. We will try a recurrent sequence processing model – it should be the perfect fit for such sequence data, precisely because it does exploit the temporal ordering of data points, unlike our first approach.

In what follows we will use the GRU layer, developed by Cho et al. in 2014. GRU layers (which stands for “gated recurrent unit”) work by leveraging the same principle as LSTM, but they are somewhat streamlined and thus cheaper to run, albeit they may not have quite as much representational power as the LSTM. This trade-off between computational expensiveness and representational power is seen everywhere in machine learning.

Lets look at our results:

```
plot(history)
```

Much better! We are able to significantly beat the random-walk baseline, which demonstrates the value of machine learning in this problem, as well as the superiority of recurrent networks compared to sequence-flattening dense networks on this type of task.

Our new validation MAE of (before we start significantly overfitting) should be much improved after de-normalization, but we probably still have a bit of margin for improvement.

Using recurrent dropout to fight overfitting

It is evident from our training and validation curves that our model is overfitting: the training and validation losses start diverging considerably after a few epochs. The classic technique for fighting this phenomenon: dropout, consisting in randomly zeroing-out input units of a layer in order to break happenstance correlations in the training data that the layer is exposed to. How to correctly apply dropout in recurrent networks, however, is not a trivial question. Following the work of Yarin Gal, we use the same dropout mask at every timestep, which allows the network to properly propagate its learning error through time.

Every recurrent layer in Keras has two dropout-related arguments: `dropout` , a float specifying the dropout rate for input units of the layer, and `recurrent_dropout` , specifying the dropout rate of the recurrent units. Let’s add dropout and recurrent dropout to our GRU layer and see how it impacts overfitting. Since networks that are regularized with dropout always take longer to fully converge, we train our network for twice as many epochs.

```
plot(history)
```

Great success; we are no longer overfitting during the first 30 epochs. However, while we have more stable evaluation scores, our best scores are not much lower than they were previously.

Stacking recurrent layers

Since we are no longer overfitting yet we seem to have hit a performance bottleneck, we should start considering increasing the capacity of our network. It is a generally a good idea to increase the capacity of your network until overfitting becomes your primary obstacle (assuming that you are already taking basic steps to mitigate overfitting, such as using dropout). As long as you are not overfitting too badly, then you are likely under-capacity.

Increasing network capacity is typically done by increasing the number of units in the layers, or adding more layers. Recurrent layer stacking is a classic way to build more powerful recurrent networks: for instance, what currently powers the Google translate algorithm is a stack of seven large LSTM layers, which is massive.

To stack recurrent layers on top of each other in Keras, all intermediate layers should return their full sequence of outputs (a 3D tensor) rather than their output at the last timestep. This is done by specifying `return_sequences = TRUE` :

Let’s take a look at our results:

```
plot(history)
```

We can see that the added layers does improve ours results by a bit, albeit not very significantly. We can draw two conclusions:

- Since we are still not overfitting, we could safely increase the size of our layers, in quest for a bit of validation loss improvement. This does have a non-negligible computational cost, though.
- Since adding a layer did not help us by a significant factor, we may be seeing diminishing returns to increasing network capacity at this point.

Using bidirectional RNNs

The last technique that we will introduce in this section is called “bidirectional RNNs”. A bidirectional RNN is common RNN variant which can offer higher performance than a regular RNN on certain tasks. It is frequently used in natural language processing.

RNNs are notably order-dependent, or time-dependent: they process the timesteps of their input sequences in order, and shuffling or reversing the timesteps can completely change the representations that the RNN will extract from the sequence. This is precisely the reason why they perform well on problems where order is meaningful, such as our temperature forecasting problem. A bidirectional RNN exploits the order-sensitivity of RNNs: it simply consists of two regular RNNs, such as the GRU or LSTM layers that you are already familiar with, each processing input sequences in one direction (chronologically and antichronologically), then merging their representations. By processing a sequence both way, a bidirectional RNN is able to catch patterns that may have been overlooked by a one-direction RNN.

Remarkably, the fact that the RNN layers in this section have processed sequences in chronological order (older timesteps first) may have been an arbitrary decision. Could the RNNs have performed well enough if they processed input sequences in antichronological order, for instance (newer timesteps first)? Let’s try this in practice and see what happens. All you need to do is write a variant of the data generator where the input sequences are reverted along the time dimension (replace the last line with `list(samples[,ncol(samples):1,], targets)`). Training the same one-GRU-layer network that you used in the first experiment in this section, you get the results shown below.

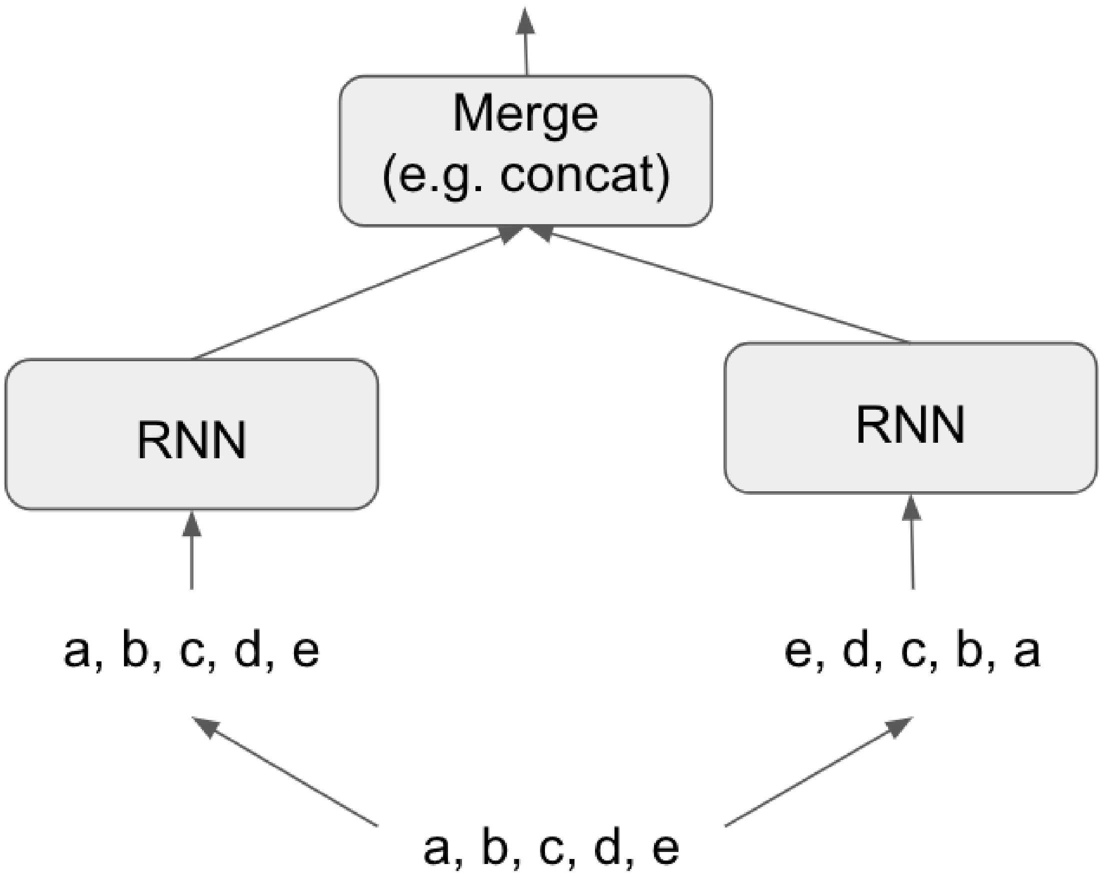
```
reverse_order_generator <- function( data, lookback, delay, min_index, max_index,
                                     shuffle = FALSE, batch_size = 128, step = 6) {
  if (is.null(max_index))
    max_index <- nrow(data) - delay - 1
  i <- min_index + lookback
  function() {
    if (shuffle) {
      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[,ncol(samples):1,], targets)
  }
}

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

val_gen_reverse = reverse_order_generator(
  data,
  lookback = lookback,
  delay = delay,
  min_index = 200001,
  max_index = 300000,
  step = step,
  batch_size = batch_size
)
```

```
plot(history)
```

So the reversed-order GRU strongly underperforms even the common-sense baseline, indicating that in our case chronological processing is very important to the success of our approach. This makes perfect sense: the underlying GRU layer will typically be better at remembering the recent past than the distant past, and naturally the more recent weather data points are more predictive than older data points in our problem (that's precisely what makes the common-sense baseline a fairly strong baseline). Thus the chronological version of the layer is bound to outperform the reversed-order version. Importantly, this is generally not true for many other problems, including natural language: intuitively, the importance of a word in understanding a sentence is not usually dependent on its position in the sentence.

Importantly, a RNN trained on reversed sequences will learn different representations than one trained on the original sequences, in much the same way that you would have quite different mental models if time flowed backwards in the real world – if you lived a life where you died on your first day and you were born on your last day. In machine learning, representations that are *different* yet *useful* are always worth exploiting, and the more they differ the better: they offer a new angle from which to look at your data, capturing aspects of the data that were missed by other approaches, and thus they can allow to boost performance on a task. This is the intuition behind "ensembling".

A bidirectional RNN exploits this idea to improve upon the performance of chronological-order RNNs: it looks at its inputs sequence both ways, obtaining potentially richer representations and capturing patterns that may have been missed by the chronological-order version alone.



bidirectional rnn

To instantiate a bidirectional RNN in Keras, you use the `bidirectional()` function, which takes a recurrent layer instance as an argument. The `bidirectional()` function creates a second, separate instance of this recurrent layer and uses one instance for processing the input sequences in chronological order and the other instance for processing the input sequences in reversed order. Now let's try the same approach on the weather prediction task:

It performs about as well as the regular GRU layer. It's easy to understand why: all of the predictive capacity must be coming from the chronological half of the network, since the anti-chronological half is known to be severely underperforming on this task (again, because the recent past matters much more than the distant past in this case).

Going even further

There are many other things we could try, in order to improve performance on the weather-forecasting problem:

- Adjust the number of units in each recurrent layer in the stacked setup (since the current choices are largely arbitrary and thus probably suboptimal)
- Adjust the learning rate used by the `RMSprop` optimizer
- Try using `layer_lstm()` instead of `layer_gru()`
- Try using a bigger densely connected regressor on top of the recurrent layers: that is, a bigger dense layer or even a stack of dense layers
- Don't forget to eventually run the best-performing models (in terms of validation MAE) on the test set to ensure that you are not overfitting the model to the validation set

As always, deep learning is more an art than a science. We can provide guidelines that tell you what is likely to work or not work on a given problem, but, ultimately, every problem is unique; you'll have to evaluate different strategies empirically. There is currently no theory that will tell you in advance precisely what you should do to optimally solve a problem. You must iterate.