LSTM time series forecasting with TensorFlow

Dennis W. Hallema

Disclaimer: Use at your own risk. No responsibility is assumed for a user's application of these materials and related materials. These materials and referenced materials do not constitute an endorsement of any kind, not of any software, data set, method nor of anything else.

Description: These materials include a demonstration of the LSTM (Long-Short Term Memory) deep learning time series forecasting model in Keras/TensorFlow.

References:

- Hallema, D. W. (2024). Interactive Workshop, Process-Agnostic Water Quality Modeling DRI-ERDC Technology Transfer. Desert Research Institute, Las Vegas, NV, June 20, 2024.
- Hallema, D. W. (2024). LSTM (Long Short-Term Memory) forecasting of reservoir inflow and water quality in critical water supply areas. USACE-ERDC Broad Agency Announcement, No. W912HZ2320025: Drivers and disturbances of reservoir water temperature and water quality, FY 24 Initiative 3: AI-based assessment of regional water quality patterns in USACE reservoirs across the western United States (Technical Report). Desert Research Institute, July 2024, Las Vegas, Nevada, 48 p.

Overview

We can use time series forecasting to predict the volume of water coming into the reservoir from upstream. A combination of weather models and distributed hydrologic models is useful for time scales shorter than two weeks, and climate models help with seasonal and annual predictions. However, for biweekly to seasonal time scales, LSTM time series forecasting is useful because it requires little data. This has several benefits:

1. Streamflow prediction and flood forecasting: An accurate prediction of water levels in rivers and reservoirs is critical for advance flood warning. In addition, this serves fish flow management. Fish flow mapping assists resource managers in linking aquaculture, fish farming and conservation policies, and flow regulation is an important element of fish flow regulation.
2. Climate impact analysis: Sea Surface Temperatures (SSTs) are used in climate modeling, but can also be linked directly to flow or stream temperature series forecasting models. This is done by incorporating area-averaged time series of SSTs into the LSTM.
3. Potential evapotranspiration (PET) forecasting. Many distributed hydrologic models, like the Storm Water Management Model (SWMM) and HEC-HMS, and linear hydrologic reservoir models can benefit from accurate evaporation data for calibration. PET co-varies with the diurnal temperature cycle, and this knowledge can help generate a PET time series forecast.

> ***NOTE:*** Forecasting refers to the estimation of future values for a time series based on historical data. Prediction refers to the estimation of unknown past or future values in general. We speak of prediction when we train-test a model on historical data.

Preparing Anaconda

TensorFlow and Keras are both written in Python, so we will need a Python distribution like Anaconda.

1. Download and install Anaconda.
    - Install for `Just Me (recommended)`
    - ***IMPORTANT:*** Check the box `Register Anaconda3 as my default Python 3.X`
    - Check the box `Clear the package cache upon completion`
    - (Check the box `Add Anaconda to path variable`)
    - (Wait several minutes for installation to complete)
2. Add Anaconda to the PATH variable. In Windows:
    - Press *Windows* key > Type `environment variables > Enter`
    - Click `Environment Variables`
    - Under User Variables, click `Path > Edit`
    - If you don't see Anaconda listed, click `New` and add a variable `C:\Users\Name\Anaconda3\Scripts` where `Name` is your Windows user profile name.

Next, to install the TensorFlow and keras packages in the Anaconda environment, proceed as follows.

1. Press the *Windows* key and type `Anaconda Navigator`

2. Launch the `PowerShell Prompt`. This opens a PowerShell session in the Anaconda base environment, indicated by the prompt `(base) PS C:\Users\Name>`.
3. At this point you can elect to create a new Anaconda environment if you like. In this guide we'll install `keras` and a full version of `tensorflow` in the Anaconda base environment:
    - `(base) C:\Users\Name> pip install tensorflow`
    - `(base) C:\Users\Name> pip install keras`
4. Let's test if Tensorflow and Keras were installed correctly:
    - Launch Python: `(base) C:\Users\Name> python`
    - Import the Tensorflow package: `>>> import tensorflow as tf`
    - Import the Keras package: `>>> import keras` If there are no errors but only some warnings, the installation was likely succesful.

Preparing R

Install the following packages using the command `install.packages()` as shown below.

```
# (Uncomment the following lines and run to install the packages)
# install.packages("dplyr")
# install.packages("readr")
# install.packages("pracma")
# install.packages("reticulate")  # Python interface
### install.packages("devtools")
```

We also need the R package `keras3` so we can use R as an interface for Keras. When prompted, select "1" to update all keras dependencies. After the installer finishes, check the package version to ensure it was installed correctly.

```
# install.packages("keras3")
# packageVersion("keras3")

# (Try the following if the Keras installation fails)
### install.packages("devtools")
### devtools::install_github("rstudio/keras")
```

Getting the data

Copy `usgs_14210000_2014_2024.csv` to the same directory as this `.Rmd` file (the R Markdown Notebook).

(If you first need to download the data, open the following link and wait for the download to complete: https://nwis.waterdata.usgs.gov/nwis/uv?cb_all_=on&cb_00010=on&format=rdb&site_no=14210000&legacy=1&period=&begin_date=2014-04-01&end_date=2024-04-01 (https://nwis.waterdata.usgs.gov/nwis/uv?cb_all_=on&cb_00010=on&format=rdb&site_no=14210000&legacy=1&period=&begin_date=2014-04-01&end_date=2024-04-01) > Right-click > Save As `usgs_14210000_1984_2024.csv`)

The data file contains a tab-separated USGS time series for the stream gage `USGS 14210000 CLACKAMAS R. near ESTACADA Oregon` covering the period 2014-4/2024-4. Variables include stream temperature, dissolved oxygen, and turbidity, among others. To view the location where these data

were collected, browse to Google Maps (or Bing Maps) and copy-paste the coordinates: `45°18'00.0"N 122°21'10.0"W`.

> ***Question:*** What do you notice about the location of the stream gage?

Open the USGS data file in Excel. It looks like this:

```
# ---------------------------------- WARNING ----------------------------------------
# Some of the data that you have obtained from this U.S. Geological Survey database
# may not have received Director's approval. Any such data values are qualified
# as provisional and are subject to revision. Provisional data are released on the
# condition that neither the USGS nor the United States Government may be held liable
# for any damages resulting from its use.
#
# Additional info: https://waterdata.usgs.gov/provisional-data-statement/
#
# File-format description:  https://help.waterdata.usgs.gov/faq/about-tab-delimited-output
# Automated-retrieval info: https://help.waterdata.usgs.gov/faq/automated-retrievals
#
# Contact:   gs-w_waterdata_support@usgs.gov
# retrieved: 2024-04-16 11:09:02 EDT       (sdww01)
#
# Data for the following 1 site(s) are contained in this file
#    USGS 14210000 CLACKAMAS RIVER AT ESTACADA, OR
# -----------------------------------------------------------------------------------
#
# Data provided for site 14210000
#            TS   parameter     statistic     Description
#        114314       00060     00003     Discharge, cubic feet per second (Mean)
#        114315       00010     00001     Temperature, water, degrees Celsius (Maximum)
#        114316       00010     00002     Temperature, water, degrees Celsius (Minimum)
#        114317       00010     00003     Temperature, water, degrees Celsius (Mean)
#        114318       00095     00001     Specific conductance, water, unfiltered, microsi
emens per centimeter at 25 degrees Celsius (Maximum)
#        114319       00095     00002     Specific conductance, water, unfiltered, microsi
emens per centimeter at 25 degrees Celsius (Minimum)
#        114320       00095     00003     Specific conductance, water, unfiltered, microsi
emens per centimeter at 25 degrees Celsius (Mean)
#        114321       00300     00001     Dissolved oxygen, water, unfiltered, milligrams
per liter (Maximum)
#        114322       00300     00002     Dissolved oxygen, water, unfiltered, milligrams
per liter (Minimum)
#        114323       00300     00003     Dissolved oxygen, water, unfiltered, milligrams
per liter (Mean)
#        114324       00400     00001     pH, water, unfiltered, field, standard units (Ma
ximum)
#        114325       00400     00002     pH, water, unfiltered, field, standard units (Mi
nimum)
#        114326       00400     00008     pH, water, unfiltered, field, standard units (Me
dian)
#        114330       00045     00006     Precipitation, total, inches (Sum), Faraday Lake
precip
#        216058       63680     00008     Turbidity, water, unfiltered, monochrome near in
fra-red LED light, 780-900 nm, detection angle 90 +-2.5 degrees, formazin nephelometric un
its (FNU) (Median)
#        216059       63680     00001     Turbidity, water, unfiltered, monochrome near in
fra-red LED light, 780-900 nm, detection angle 90 +-2.5 degrees, formazin nephelometric un
its (FNU) (Maximum)
#        216060       63680     00002     Turbidity, water, unfiltered, monochrome near in
```

```
fra-red LED light, 780-900 nm, detection angle 90 +-2.5 degrees, formazin nephelometric un
its (FNU) (Minimum)
#        332869       32321     00001     Phycocyanin relative fluorescence (fPC), water,
in situ, relative fluorescence units (RFU) (Maximum)
#        332870       32321     00002     Phycocyanin relative fluorescence (fPC), water,
in situ, relative fluorescence units (RFU) (Minimum)
#        332871       32321     00003     Phycocyanin relative fluorescence (fPC), water,
in situ, relative fluorescence units (RFU) (Mean)
#        332873       32315     00001     Chlorophyll relative fluorescence (fChl), water,
in situ, relative fluorescence units (RFU) (Maximum)
#        332874       32315     00002     Chlorophyll relative fluorescence (fChl), water,
in situ, relative fluorescence units (RFU) (Minimum)
#        332875       32315     00003     Chlorophyll relative fluorescence (fChl), water,
in situ, relative fluorescence units (RFU) (Mean)
#
# Data-value qualification codes included in this output:
#
#     A  Approved for publication -- Processing and review completed.
#     P  Provisional data subject to revision.
#     <  Actual value is known to be less than reported value.
#     e  Value has been estimated.
#     4  Statistic computed from less than expected number of instantaneous values for the
period
#
agency_cd   site_no datetime    114314_00060_00003  114314_00060_00003_cd   114315_00010_0
0001  114315_00010_00001_cd   114316_00010_00002  114316_00010_00002_cd   114317_00010_000
03  114317_00010_00003_cd   114318_00095_00001  114318_00095_00001_cd   114319_00095_00002
114319_00095_00002_cd   114320_00095_00003  114320_00095_00003_cd   114321_00300_00001  11
4321_00300_00001_cd   114322_00300_00002  114322_00300_00002_cd   114323_00300_00003  1143
23_00300_00003_cd   114324_00400_00001  114324_00400_00001_cd   114325_00400_00002  114325
_00400_00002_cd   114326_00400_00008  114326_00400_00008_cd   114330_00045_00006  114330_0
0045_00006_cd   216058_63680_00008  216058_63680_00008_cd   216059_63680_00001  216059_636
80_00001_cd   216060_63680_00002  216060_63680_00002_cd   332869_32321_00001  332869_32321
_00001_cd   332870_32321_00002  332870_32321_00002_cd   332871_32321_00003  332871_32321_0
0003_cd   332873_32315_00001  332873_32315_00001_cd   332874_32315_00002  332874_32315_000
02_cd   332875_32315_00003  332875_32315_00003_cd
5s  15s 20d 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10
s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s 14n 10s
14n 10s 14n 10s
USGS    14210000    2014-04-01  5740    A   6.4 A   6.0 A   6.2 A   38  A   36  A   37  A
12.8    A   12.6    A   12.8    A   7.4 A   7.4 A   7.4 A           4.3 A   6.5 A   3.4 A
USGS    14210000    2014-04-02  5130    A   6.6 A   6.3 A   6.4 A   38  A   38  A   38  A
12.6    A   12.3    A   12.5    A   7.5 A   7.4 A   7.4 A           3.7 A   4.3 A   3.1 A
USGS    14210000    2014-04-03  4530    A   7.0 A   6.2 A   6.5 A   39  A   38  A   39  A
12.3    A   12.1    A   12.2    A   7.5 A   7.4 A   7.4 A           3.1 A   6.9 A   2.3 A
USGS    14210000    2014-04-04  4250    A   7.0 A   6.6 A   6.8 A   40  A   39  A   40  A
12.2    A   12.0    A   12.1    A   7.5 A   7.4 A   7.4 A           2.5 A   3.0 A   2.1 A
```

…etc for all days in the data set.

Importing the data

As shown above, the data comes with multiple lines of comments preceded by a `#`, and a header line. We will use the `readLines` function to read in the whole file line-by-line, and then `read_table` (`readr` package), which allows us to automatically organize the raw data into columns.

To ensure R can find our data, copy the file `usgs_14210000_1984_2024.csv` in the same folder as the R Markdown notebook. Next, return to RStudio and select the `Session` menu > `Set working directory` > `To source file location`. If we attempt to open a file, R will now look in the directory where the source file is also located.

We will now import the data. Note that R may report some parsing failures, but that is not a problem.

```
# Load packages
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(readr)

# Read raw data file
file <- "usgs_14210000_1984_2024.csv"
cleanlines = readLines(file)

# Remove comments and first header line
for(i in c("#","5s")){ cleanlines <- sub(paste0(i, ".*"), "", cleanlines) }

# Automatically organize the raw data into columns
datraw <- read_table(cleanlines)
```

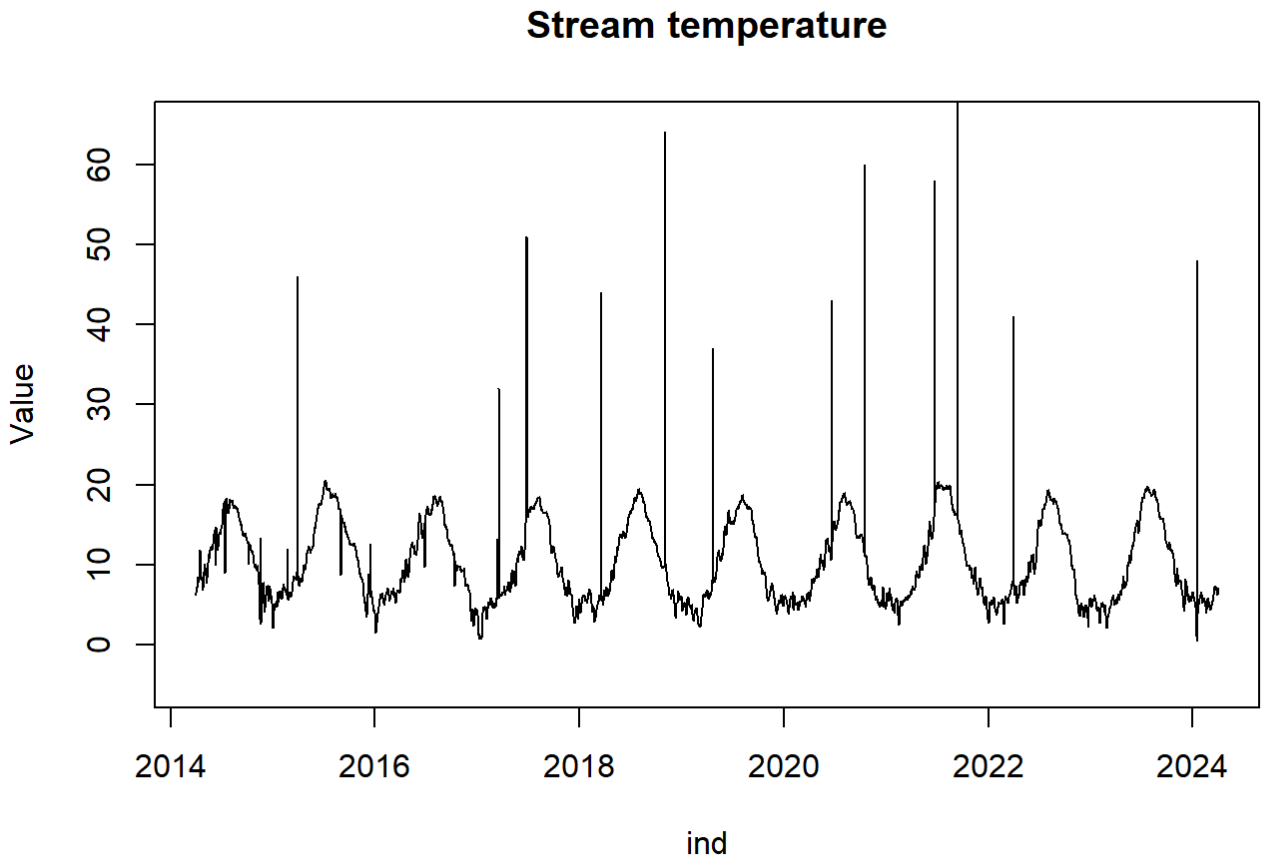
```
## Warning: 3657 parsing failures.
## row col   expected     actual         file
##   1  -- 49 columns 36 columns literal data
##   2  -- 49 columns 36 columns literal data
##   3  -- 49 columns 36 columns literal data
##   4  -- 49 columns 36 columns literal data
##   5  -- 49 columns 36 columns literal data
## ... ... .......... .......... ............
## See problems(...) for more details.
```

Copy the column with the label `Temperature, water, degrees Celsius (Mean)`. According to the comment lines at the beginning of the data file (the lines marked with `#`), we need the column with the index `114317_00010_00003`.

```
# Copy the column of interest
dat <- data.frame(ind = datraw$datetime, Value = as.numeric(datraw$`114317_00010_00003`))

# Inspect the first few lines
head(dat)
```

```
##          ind Value
## 1 2014-04-01   6.2
## 2 2014-04-02   6.4
## 3 2014-04-03   6.5
## 4 2014-04-04   6.8
## 5 2014-04-05   6.9
## 6 2014-04-06   7.2
```

```
# Visualize the data
plot(dat, main = "Stream temperature", type = "l", ylim = c(-5,65))
```



Question: What stands out in this plotted time series?

Robust outlier filtering

Outliers can represent true values or result from error (observational error, data error, etc.) To separate between true extremes from extreme values due to error, we apply a Hampel filter. The Hampel filter computes the median value for a running window between -k and k time steps away of each point, and determines the median of absolute deviations (MAD) from this running median. If a data point lies more than t0 x MAD away from the running median, we may presume that its value is not a true value. We replace those with the running median. Let's identify the erroneous extreme values.

```
# Load the PRACMA package
library(pracma)

# non NA values index
ValInd <- which(!is.na(dat$Value))

# Use the Hampel filter (mean absolute deviation from median) to find the rows that have o
utliers
OutlierInd <- ValInd[pracma::hampel(dat$Value[ValInd],k=7, t0=7)$ind]
# dat$Value[OutlierInd]

# Number of outliers detected
nrow(dat[OutlierInd,])
```

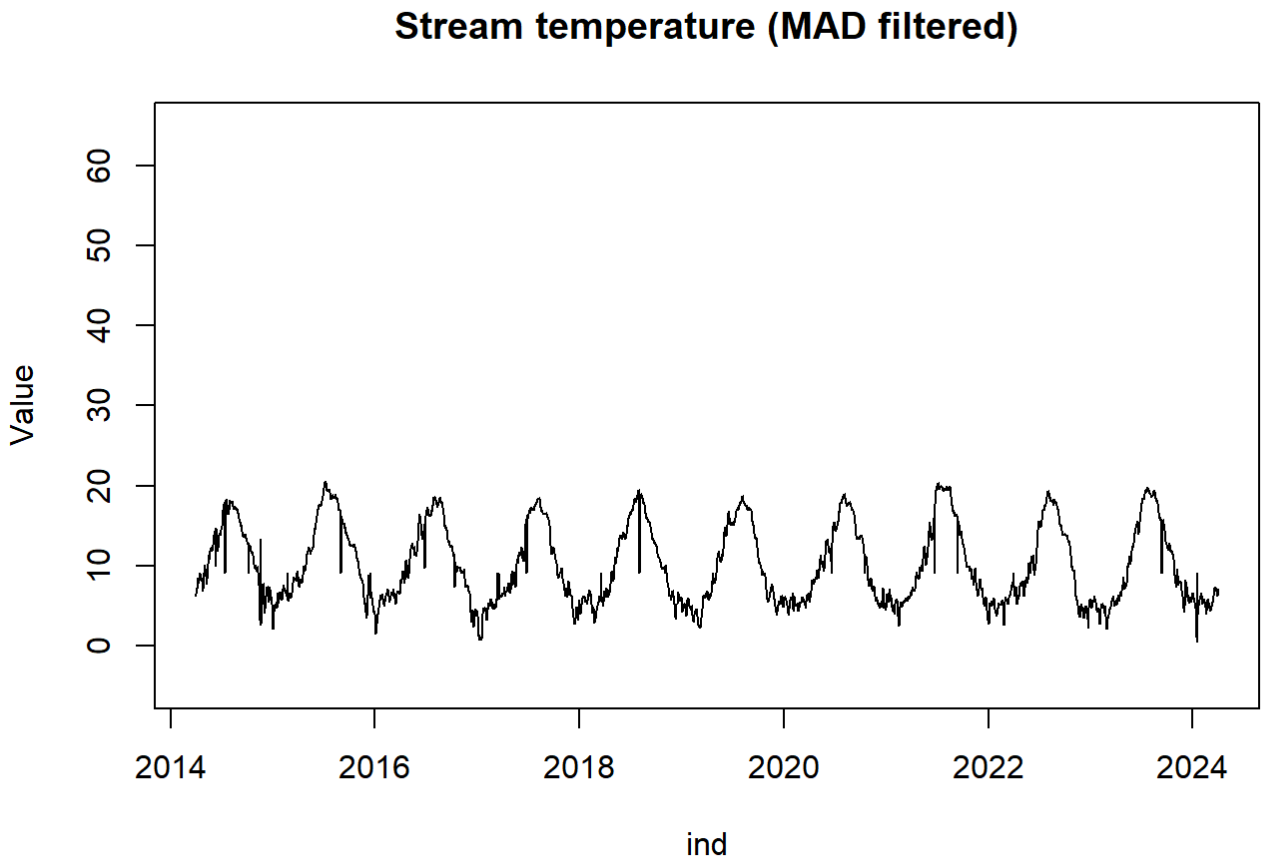
```
## [1] 23
```

```
# View all outliers
dat[OutlierInd,]
```

```
##             ind Value
## 17   2014-04-17  11.8
## 106  2014-07-15   9.0
## 190  2014-10-07  10.1
## 330  2015-02-24  12.0
## 365  2015-03-31  46.0
## 520  2015-09-02   8.7
## 625  2015-12-16  12.6
## 954  2016-11-09   9.3
## 1079 2017-03-14  13.2
## 1084 2017-03-19  32.0
## 1183 2017-06-26  51.0
## 1450 2018-03-20  44.0
## 1587 2018-08-04  18.8
## 1677 2018-11-02  64.0
## 1849 2019-04-23  37.0
## 2272 2020-06-19  43.0
## 2391 2020-10-16  60.0
## 2640 2021-06-22  58.0
## 2724 2021-09-14  71.0
## 2922 2022-03-31  41.0
## 3451 2023-09-11  15.8
## 3453 2023-09-13  15.8
## 3579 2024-01-17  48.0
```

Next, we remove the idenfied outliers and visualize the resulting time series.

```
# Replace outliers
for (i in 1:length(OutlierInd)) {
  dat$Value[OutlierInd] <- median(dat$Value, na.rm=T)
}

# Plot the time series again
plot(dat, main = "Stream temperature (MAD filtered)", type = "l", ylim = c(-5,65))
```



> ***Question:*** What Hampel window size did we use here? And what filter threshold? Tip: Open the help documentation for the Hampel function using `?hampel`.

> ***Question:*** What is the advantage of using a Hampel median absolute deviation (MAD) filter to remove outliers as opposed to filters based on a running mean?

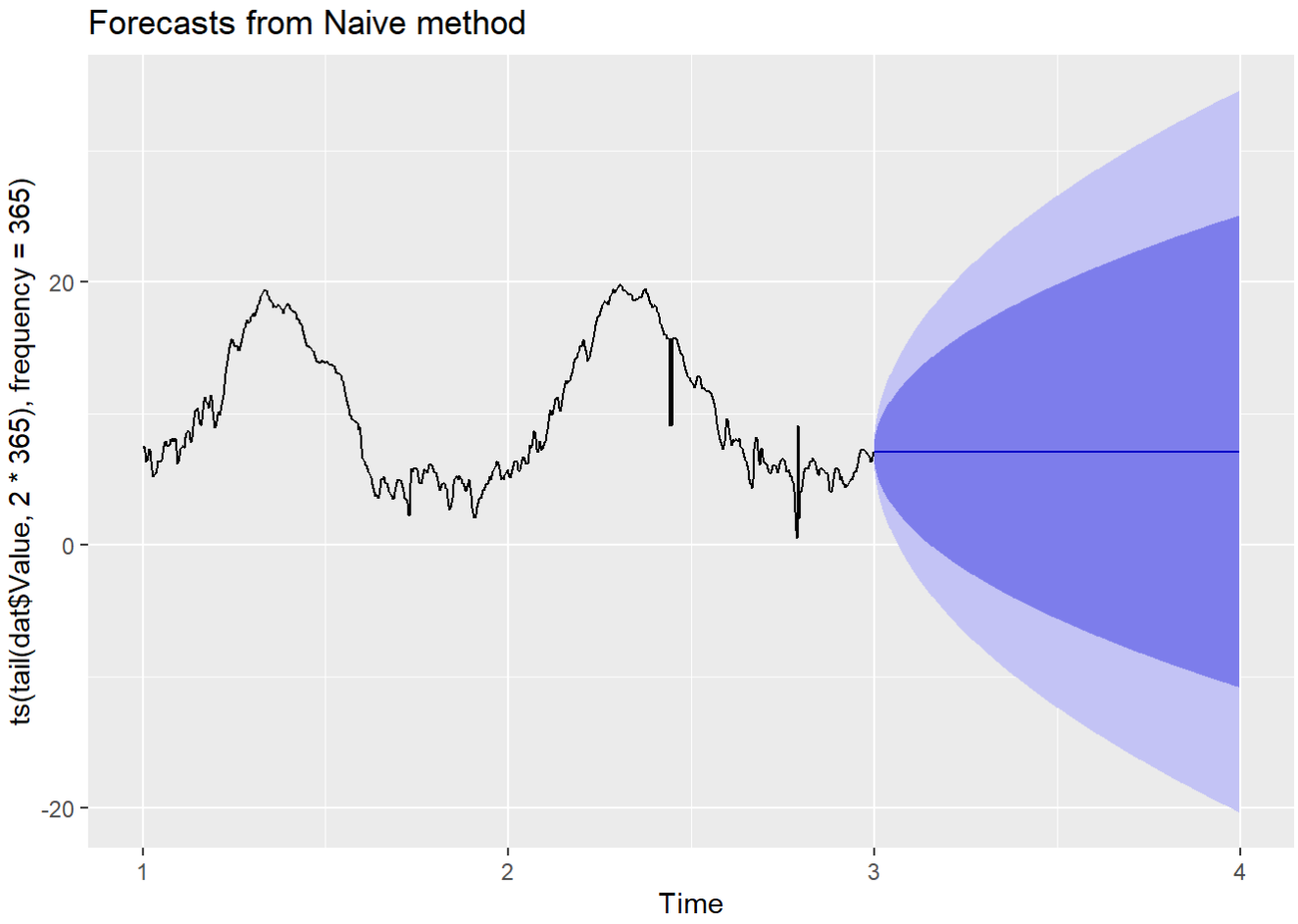
Benchmark forecasting the time series data

We will get to LSTM deep learning forecasting in a bit. To understand the advantage of LSTMs, it can help to look at other forecasting methods (benchmarking) first. The simplest benchmark is based on the assumption that nothing changes, i.e. we compute an expected value from the mean of all the previous values. This yields a rough estimate of what we can expect, and we can add a confidence interval, which we expect to increase as a function of time.

```
# Load the forecast package
library(forecast)
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

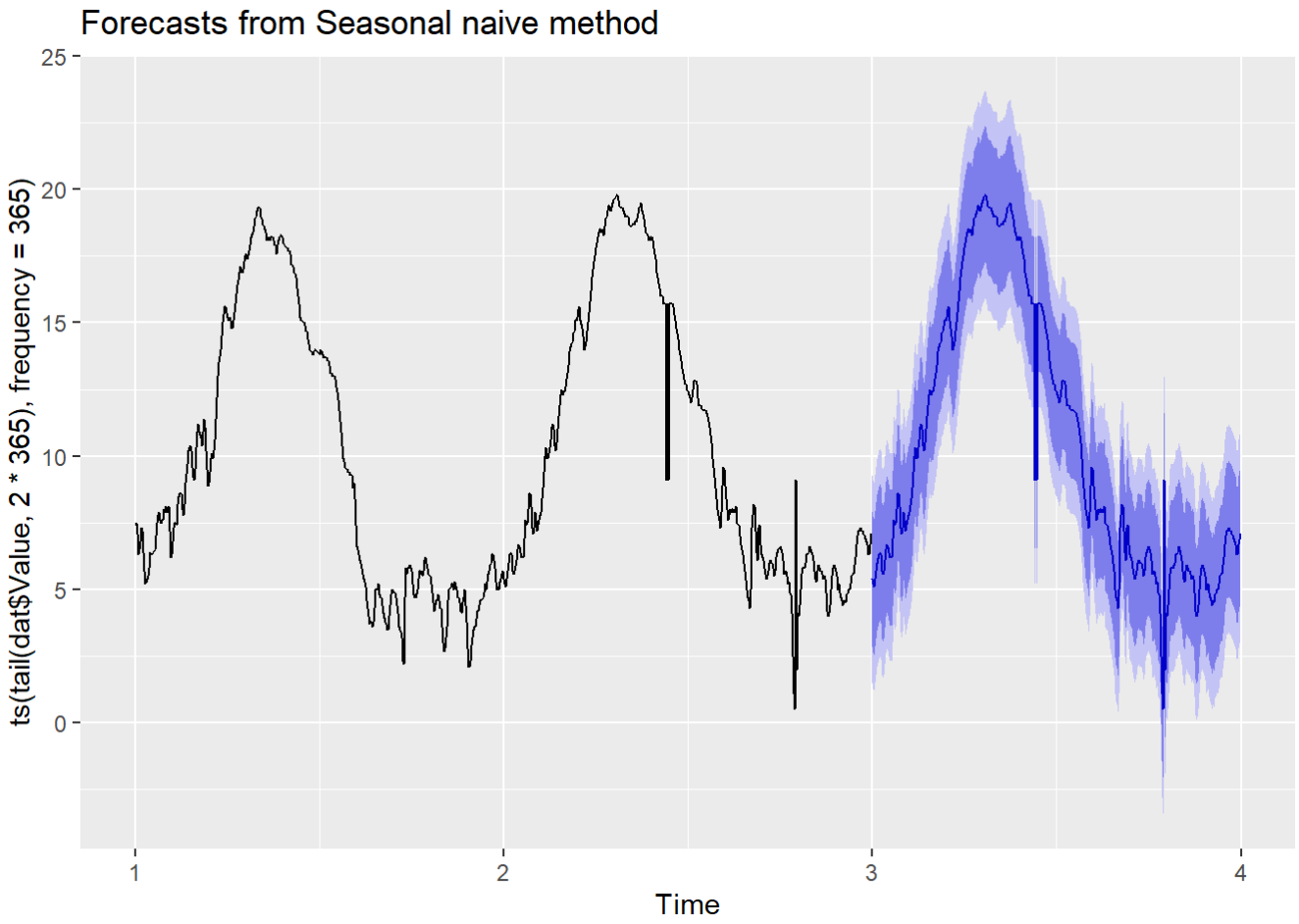
```
# Naive forecast
ndat <- naive(ts(tail(dat$Value,2*365), frequency = 365), h = 365)

# Plot and summarize the forecasts
autoplot(ndat)
```



> ***Question:*** What does the naive function do? Tip: use the command `?naive`.

Now let's add a seasonal component to the forecasting model.

```
# Seasonal naive forecast
sndat <- snaive(ts(tail(dat$Value,2*365), frequency = 365), h = 365)

# Plot and summarize the forecasts
autoplot(sndat)
```



> ***Question:*** How does the seasonal naive streamflow temperature forecast method compare to the naive forecast method?

ARIMA models

Naive models are Autoregressive Integrated Moving Average (ARIMA) models. ARIMA uses the lag and shift in time series data to predict future patterns. The model was first proposed in 1976 and has gained popularity in economic forecasting and short-term business performance forecasting. To understand how ARIMA works, we need to discuss its predecessors because ARIMA combines components of these earlier models.

- Autoregressive (AR) models: multiple regression using lagged observations as predictors.
- Moving Average (MA) models: multiple regression using lagged residuals (errors) as predictors.
- Autoregressive Moving Average (ARMA) models: multiple regression using both lagged observations and lagged residuals as predictors. Accepts only stationary data.
- Autoregressive Integrated Moving Average (ARIMA{p,d,q}) models: This is generalization of the ARMA model with differencing. This allows ARIMA to account for non-stationarity in the data. The ARIMA parameters (p,d,q) are non-negative integers describing the autoregressive, differencing, and moving average terms:
    - p: order of the autoregressive (AR) model, i.e. the number of lagged observations
    - d: degree of differencing (I), i.e. the number of differenced observations
    - q: order of the moving-average (MA) (or seasonal) model In addition to (p,d,q), ARIMA models can also include intercepts or constants (c), in some cases referred to as the “drift”.

These are some common and less common ARIMA model configurations:

- ARIMA(0,0,0): white noise model
- ARIMA(0,0,1): first-order moving average (MA) model. Same as MA(1)
- ARIMA(0,1,0): random walk (if this model uses a constant c it describes a random walk with drift). Same as I(1). This is essentially a stochastic process.
- ARIMA(0,1,1) without constant: simple exponential smoothing
- ARIMA(0,1,1) with constant: simple exponential smoothing with growth
- ARIMA(0,1,2): Damped Holt's model
- ARIMA(0,2,1) or (0,2,2) without constant: linear exponential smoothing
- ARIMA(0,2,2): double exponential smoothing
- ARIMA(0,3,3): triple exponential smoothing
- ARIMA(1,0,0): first-order autoregressive model. If the series is stationary and autocorrelated, and can be predicted as a multiple of the last value (plus constant). Same as AR(1)
- ARIMA(1,1,0): differenced first-order autoregressive model. Used if the errors of the random walk model are autocorrelated but without seasonality.
- ARIMA(1,1,2) without constant: damped-trend linear exponential smoothing
- ARIMA(2,1,1): second order autoregressive, first order moving average, differenced once
  auto.arima(): The auto.arima() function in the forecast package for R can automatically select the best ARIMA model. It conducts a search over possible models within a set of constraints, and returns the best ARIMA model according to the AICC (or AIC or BIC) value.

> ***Question:*** What type of ARIMA(p,d,q) configuration describes a naive model? And a seasonal naive model? Tip: `?naive`

TensorFlow LSTM - Importing keras into R

Now we arrive at the deep learning model. First, we will setup R so we can use it as a Python interface.

```
# Set the Python path to Anaconda Python
Sys.setenv(RETICULATE_PYTHON = paste0("C:/Users/", Sys.getenv("USERNAME"), "/AppData/Loca
l/anaconda3/python.exe"))
Sys.setenv(PYTHON = paste0("C:/Users/", Sys.getenv("USERNAME"), "/AppData/Local/anaconda3/
python.exe"))
library(reticulate)
reticulate::py_config()
```

```
## python:         C:/Users/dhallema/AppData/Local/anaconda3/python.exe
## libpython:      C:/Users/dhallema/AppData/Local/anaconda3/python312.dll
## pythonhome:     C:/Users/dhallema/AppData/Local/anaconda3
## version:        3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MS
C v.1929 64 bit (AMD64)]
## Architecture:   64bit
## numpy:          C:/Users/dhallema/AppData/Local/anaconda3/Lib/site-packages/numpy
## numpy_version:  1.26.4
##
## NOTE: Python version was forced by RETICULATE_PYTHON
```

```
# Import keras into Python
reticulate::import("keras")
```

```
## Module(keras)
```

```
library(keras3)
```

LSTM data preparation

Next, let's collect the values of interest and impute the NAs. Unlike other models, neural networks are designed to demote any data points without predictive power or relation to adjacent data points. In data sets like this where `0` is not a common value in the data, the best practice is to impute all NAs with `0`.

```
# Load the reticulate and keras packages
library(reticulate)
library(keras3)

# Generate model input
ts <- data.frame(
  date = as.character(dat$ind),
  val = as.numeric(dat$Value)
  )

# Impute NAs
ts$val[is.na(ts$val)] <- 0

# Group by date
ts_grouped <- ts %>% group_by(date) %>%
  summarise(val=mean(val)) %>%
  as.data.frame()
dt <- ts_grouped

# Center and scale data
values <- data.frame(scale(dt$val))
```

In the last step above, we also center and scale the data. To train the LSTM, we will need to select the data we will use for training. We'll set a small portion of the data aside so we can test the performance of the model.

```
# Function to lag and split data into train and test
lagsplit_data <- function(tseries, test_size = 0.2, lookback = 30) {
  data_raw <- as.matrix(tseries) # convert to matrix
  data <- array(dim = c(0, lookback, ncol(data_raw)))

  # Lag the data lookback times and arrange into columns
  for (i in 1:(nrow(data_raw) - lookback)) {
    data <- rbind(data, data_raw[i:(i + lookback - 1), ])
  }

  test_set_size <- round(test_size * nrow(data))
  train_set_size <- nrow(data) - test_set_size

  x_train <- data[1:train_set_size, 1:(lookback - 1), drop = FALSE]
  y_train <- data[1:train_set_size, lookback, drop = FALSE]

  x_test <- data[(train_set_size + 1):nrow(data), 1:(lookback - 1), drop = FALSE]
  y_test <- data[(train_set_size + 1):nrow(data), lookback, drop = FALSE]

  cat(paste('x_train.shape = ', dim(x_train), '\n'))
  cat(paste('y_train.shape = ', dim(y_train), '\n'))
  cat(paste('x_test.shape = ', dim(x_test), '\n'))
  cat(paste('y_test.shape = ', dim(y_test), '\n'))

  # Return a list with train and test datasets lagged for each lookback period
  return(list(x_train = x_train, y_train = y_train,
              x_test = x_test, y_test = y_test))
}
```

```
# Lag and split scaled data into train and test
test_size <- 0.2
lookback <- 60

values_split <- lagsplit_data(values, test_size, lookback)
```

```
## x_train.shape =  2875
##  x_train.shape =  59
## y_train.shape =  2875
##  y_train.shape =  1
## x_test.shape =  719
##  x_test.shape =  59
## y_test.shape =  719
##  y_test.shape =  1
```

```
x_train <- values_split$x_train
y_train <- values_split$y_train
x_test <- values_split$x_test
y_test <- values_split$y_test
```

Building the LSTM

```
# Hyperparameters
input_dim <- 1
hidden_dim <- 60 # Units in LSTM (cf. nodes in NN)
# hidden_dim <- round(.5*lookback)
num_layers <- 2
output_dim <- 1
num_epochs <- 7 #50

# Reshape the train and test data into 3D tensor shape
x_train <- array_reshape(x_train, c(dim(x_train)[1], lookback-1, input_dim))
x_test <- array_reshape(x_test, c(dim(x_test)[1], lookback-1, input_dim))

# Define the LSTM model using Keras
model <- keras_model_sequential() %>%
  layer_lstm(units = hidden_dim, return_sequences = TRUE, input_shape = c(lookback-1, inpu
t_dim)) %>%
  layer_lstm(units = hidden_dim) %>%
  layer_dense(units = output_dim)

# Compile the model using the mse loss and the Adam optimizer
model %>% compile(loss = "mean_squared_error", optimizer = optimizer_adam(learning_rate =
0.01))
```

> ***QUESTION :*** What does the learning rate do?

Training the LSTM

```
# Train the model

# Include all data for validation
fitted <- model %>% fit(x_train, y_train, epochs = num_epochs, batch_size = 16, validation
_data = list(x_test, y_test))
```

```
## Epoch 1/7
## 180/180 - 7s - 37ms/step - loss: 0.0633 - val_loss: 0.0227
## Epoch 2/7
## 180/180 - 4s - 22ms/step - loss: 0.0261 - val_loss: 0.0223
## Epoch 3/7
## 180/180 - 4s - 24ms/step - loss: 0.0249 - val_loss: 0.0333
## Epoch 4/7
## 180/180 - 3s - 19ms/step - loss: 0.0235 - val_loss: 0.0173
## Epoch 5/7
## 180/180 - 3s - 18ms/step - loss: 0.0227 - val_loss: 0.0183
## Epoch 6/7
## 180/180 - 4s - 21ms/step - loss: 0.0213 - val_loss: 0.0367
## Epoch 7/7
## 180/180 - 4s - 23ms/step - loss: 0.0228 - val_loss: 0.0182
```

```
# Extract predictions from the estimated model
y_train_pred <- model %>% predict(x_train)
```

```
## 90/90 - 1s - 14ms/step
```

```
y_test_pred <- model %>% predict(x_test)
```

```
## 23/23 - 0s - 10ms/step
```

```
# Rescale the predictions and original values
y_train_pred_orig <- y_train_pred * sd(dt$val) + mean(dt$val)
y_train_orig <- y_train * sd(dt$val) + mean(dt$val)
y_test_pred_orig <- y_test_pred * sd(dt$val) + mean(dt$val)
y_test_orig <- y_test * sd(dt$val) + mean(dt$val)
```

Compute performance metrics

```
# Train MSE and ME
# mse <- fitted$metrics$val_loss[length(fitted$metrics$val_loss)]
rmse <- (mean((y_train_pred_orig - y_train_orig)^2))^.5
me <- mean(y_train_pred_orig - y_train_orig)
summary(y_train_orig)
```

```
##        V1
##  Min.   : 0.00
##  1st Qu.: 6.00
##  Median : 9.20
##  Mean   :10.25
##  3rd Qu.:14.50
##  Max.   :20.60
```

```
# Train Peak error
y_train_orig_p90 <- quantile(y_train_orig, .90)[[1]]
y_train_orig_p90
```

```
## [1] 17.6
```

```
rmse_p90 <- (mean((y_train_orig[y_train_orig >= y_train_orig_p90] - y_train_pred_orig[y_tr
ain_orig >= y_train_orig_p90])^2))^.5
me_p90 <- mean( y_train_orig[y_train_orig >= y_train_orig_p90] - y_train_pred_orig[y_train
_orig >= y_train_orig_p90])
mpe_p90 <- 100*mean( (y_train_orig[y_train_orig >= y_train_orig_p90] - y_train_pred_orig[y
_train_orig >= y_train_orig_p90])/y_train_orig[y_train_orig >= y_train_orig_p90])

# Test MSE and ME
rmse_test <- (mean((y_test_pred_orig - y_test_orig)^2))^.5
me_test <- mean(y_test_pred_orig - y_test_orig)

# Test Peak error
y_test_orig_p90 <- quantile(y_test_orig, .90)[[1]]
y_test_orig_p90
```

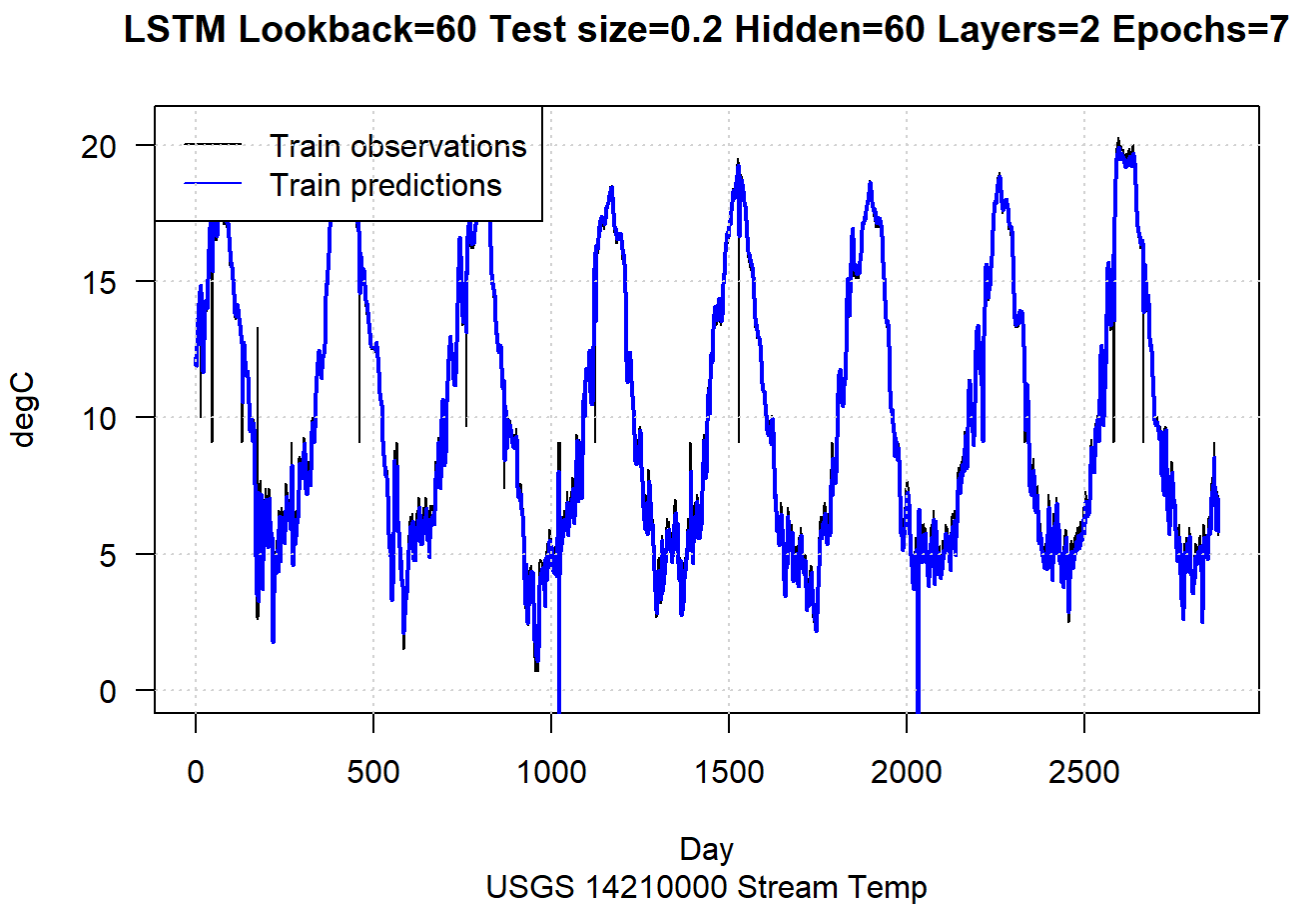
```
## [1] 18.2
```

```
rmse_test_p90 <- (mean(( y_test_orig[y_test_orig >= y_test_orig_p90] - y_test_pred_orig[y_
test_orig >= y_test_orig_p90])^2))^.5
me_test_p90 <- mean(y_test_orig[y_test_orig >= y_test_orig_p90] - y_test_pred_orig[y_test_
orig >= y_test_orig_p90] )
mpe_test_p90 <- 100*mean( (y_test_orig[y_test_orig >= y_test_orig_p90] - y_test_pred_orig
[y_test_orig >= y_test_orig_p90])/ y_test_orig[y_test_orig >= y_test_orig_p90])
summary(y_test_orig)
```

```
##        V1
##  Min.   : 0.5
##  1st Qu.: 5.6
##  Median : 8.1
##  Mean   :10.0
##  3rd Qu.:14.7
##  Max.   :19.8
```

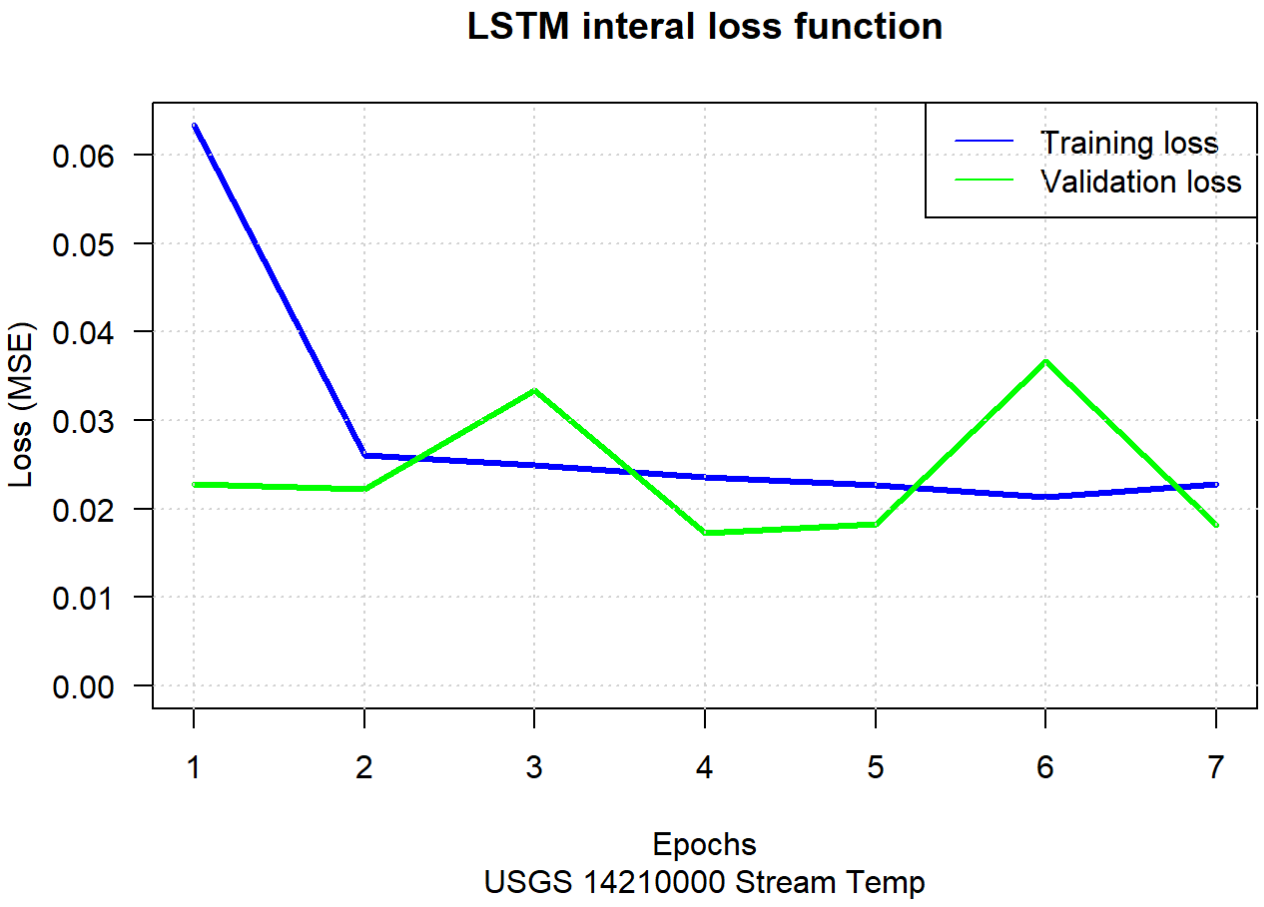
Plot the LSTM train observations and train predictions

```
pl_title <- "USGS 14210000 Stream Temp"
ylabel <- "degC"

# Set up the plot layout
par(mfrow = c(1, 1))
options(repr.plot.width=15, repr.plot.height=5)

# Plot the train observations and train predictions
plot(y_train_orig, type = "l", main = paste0("LSTM", " Lookback=",lookback," Test size=",t
est_size," Hidden=",hidden_dim," Layers=",num_layers," Epochs=",num_epochs),
     sub = pl_title, col = "black", xlab = "Day", ylab = ylabel, lwd = 1, las=1)
lines(y_train_pred_orig, col = "blue", lwd = 2)
legend(x = "topleft", legend = c("Train observations", "Train predictions"), col = c("blac
k", "blue"), lwd = 1)
grid()
```



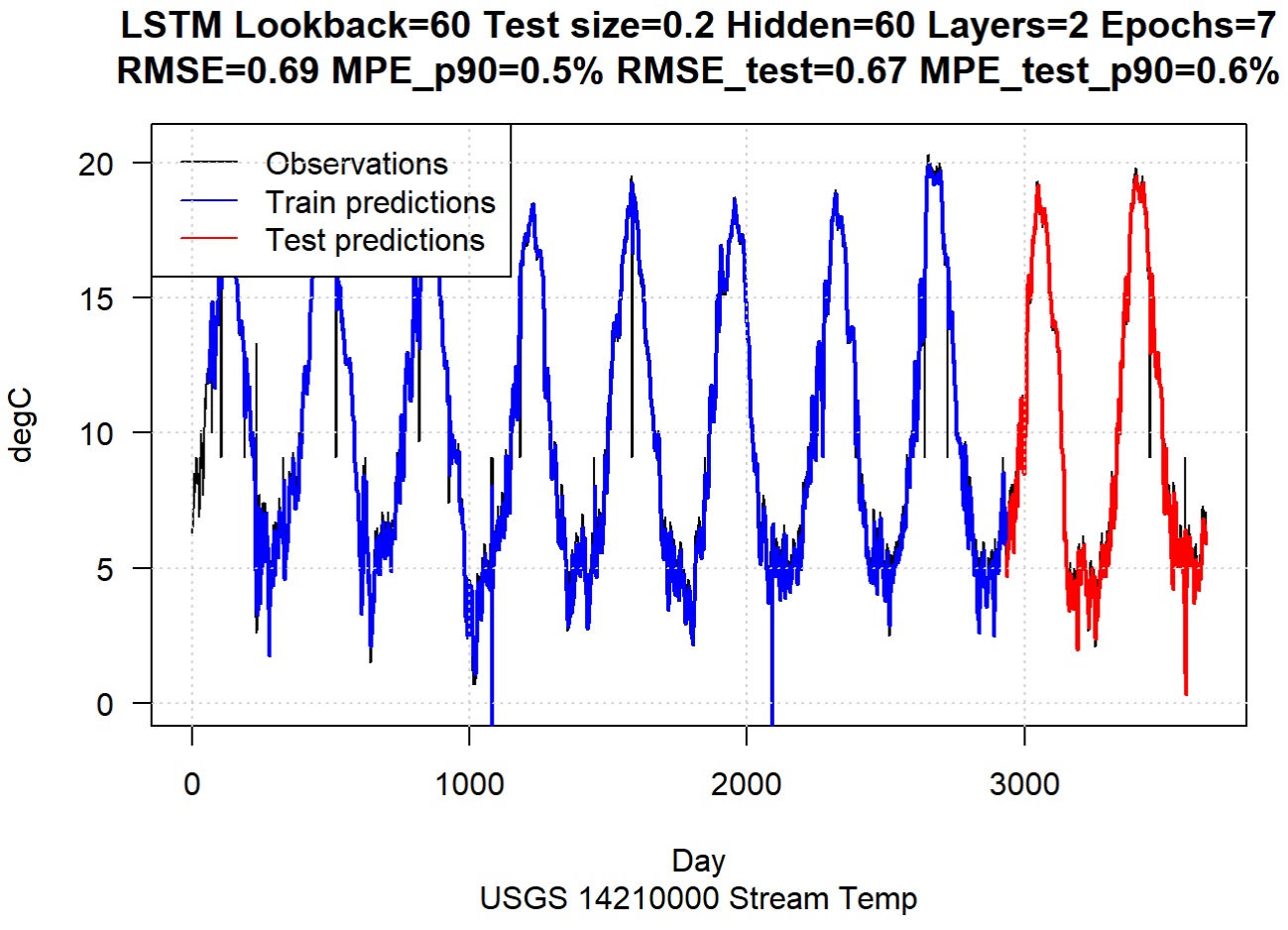
LSTM loss functions

```
# Plot the train and test loss
plot(fitted$metrics$loss, type = "l", main = "LSTM interal loss function", xlab = "Epoch
s", ylab = "Loss (MSE)", sub = pl_title, col = "blue",lwd=3 , las=1, ylim = c(0,max(fitted
$metrics$loss, fitted$metrics$val_loss)))
lines(fitted$metrics$val_loss, type="l", col = "green",lwd=3 , las=1)
legend(x = "topright", legend = c("Training loss", "Validation loss"), col = c("blue", "gr
een"), lwd = 1)
grid()
```



Plot all observations, LSTM train predictions and testing predictions

```
# Appending the testing predictions to the train predictions
y_train_pred_orig_shifted <- c(rep(NA, lookback), y_train_pred_orig[,1])
shift <- lookback + length(y_train_pred_orig)
y_test_pred_orig_shifted <- c(rep(NA, shift), y_test_pred_orig[,1])
# y_train_pred_orig_shifted <- c(rep(NA, lookback), y_train_pred_orig)
# shift <- lookback + length(y_train_pred_orig)
# y_test_pred_orig_shifted <- c(rep(NA, shift), y_test_pred_orig)

# Plot all observations, train predictions and testing predictions
options(repr.plot.width = 12, repr.plot.height = 8)
plot(dt$val, type = "l", main = paste0("LSTM", " Lookback=",lookback," Test size=",test_si
ze," Hidden=",hidden_dim," Layers=",num_layers," Epochs=",num_epochs, "\nRMSE=", round(rms
e,2), " MPE_p90=", round(mpe_p90,1), "% RMSE_test=", round(rmse_test,2), " MPE_test_p90=",
round(mpe_test_p90,1),"%"),
     sub = pl_title,col = "black", xlab = "Day", ylab = ylabel, lwd = 1, las=1)
lines(y_train_pred_orig_shifted, col = "blue",lwd = 2)
lines(y_test_pred_orig_shifted, col = "red",lwd = 2)
legend(x = "topleft", legend = c("Observations", "Train predictions","Test predictions"),
col = c("black","blue" ,"red"), lwd = 1)
grid()
```



Some of the unique features of LSTM neural networks are:

- Long term memory: LSTMs can remember hidden states and register dependencies over long sequences of time
- No vanishing gradient problem: During backpropagation, output is multiplied by the “forget” gate, not by the weights
- Not sensitive to gaps in the data

> ***QUESTION :*** When would you prefer LSTMs for time series forecasting vs a benchmark method?

EOF