Time Series Forecasting using LSTM in R

2018-04-05 · 1541 WORDS · 8 MINUTE READ DEEP LEARNING
KERAS · R · TENSORFLOW

In mid 2017, R launched package *Keras*, a comprehensive library which runs on top of Tensorflow, with both CPU and GPU capabilities. I highlighted its implementation here. In this blog I will demonstrate how we can implement time series forecasting using LSTM in R.

Brief Introduction

Time series involves data collected sequentially in time. I denote univariate data by $x_t \in \mathbb{R}$ where $t \in \mathcal{T}$ is the time indexing when the data was observed. The time t can be discrete in which case $\mathcal{T} = \mathbb{Z}$ or continuous with $\mathcal{T} = \mathbb{R}$. For simplicity of the analysis we will consider only discrete time series.

Long Short Term Memory (LSTM) networks are special kind of Recurrent Neural Network (RNN) that are capable of learning long-term dependencies. In regular RNN small weights are multiplied over and over through several time steps and the gradients diminish asymptotically to zero- a condition known as vanishing gradient problem.

LSTM netowrk typically consists of memory blocks, referred to as cells, connected through layers. The information in the cells is cointained in cell state C_t and hidden state h_t and it is regulated by mechanisms, known as gates, through *sigmoid* and *tanh* activation functions.

The sigmoid function/layer outputs numbers between 0 and 1 with 0 indicating *Nothing goes through* and 1 implying *Everything goes through*. LSTM, therefore, have the ability to, conditionally, add or delete information from the cell state.

In general, the gates take in, as input, the hidden states from previous time step h_{t-1} and the current input x_t and multiply them pointwise by weight matrices, W, and a bias b is added to the product.

Three main gates:

1. Forget gate:
   - This determine what information will be deleted from the cell state.
   - The output is a number between 0 and 1 with 0 meaning *delete all* and 1 implying *remember all*

$$f_t = \sigma\big(W_f[h_{t-1}, x_t] + b_f\big)$$

2. Input gate:
   - In this step, the *tahn* activation layer create a vector of potential canditate as follows:

$$\hat{C}_t = \tanh\big(W_c[h_{t-1}, x_t] + b_c)$$

   - The sigmoid layer creates an update filter as follows:

$$U_t = \sigma\big(W_u[h_{t-1}, x_t] + b_u\big)$$

   - Next, the old cell state C_{t-1} is updated as follows:

$$C_t = f_t * C_{t-1} + U_t * \hat{C}_t$$

3. Output gate:
   - In this step, the sigmoid layer filters the cell state that is going to output.

$$O_t = \sigma\big(W_o[h_{t-1}, x_t] + b_o)$$

   - The cell state C_t is then passed through *tanh* function to scale the values to the range [-1, 1].
   - Finally, the scaled cell state is multiplied by the filtered output to obtain the hidden state h_t to be passed on to the next cell:

$$h_t = O_t * tanh(C_t)$$

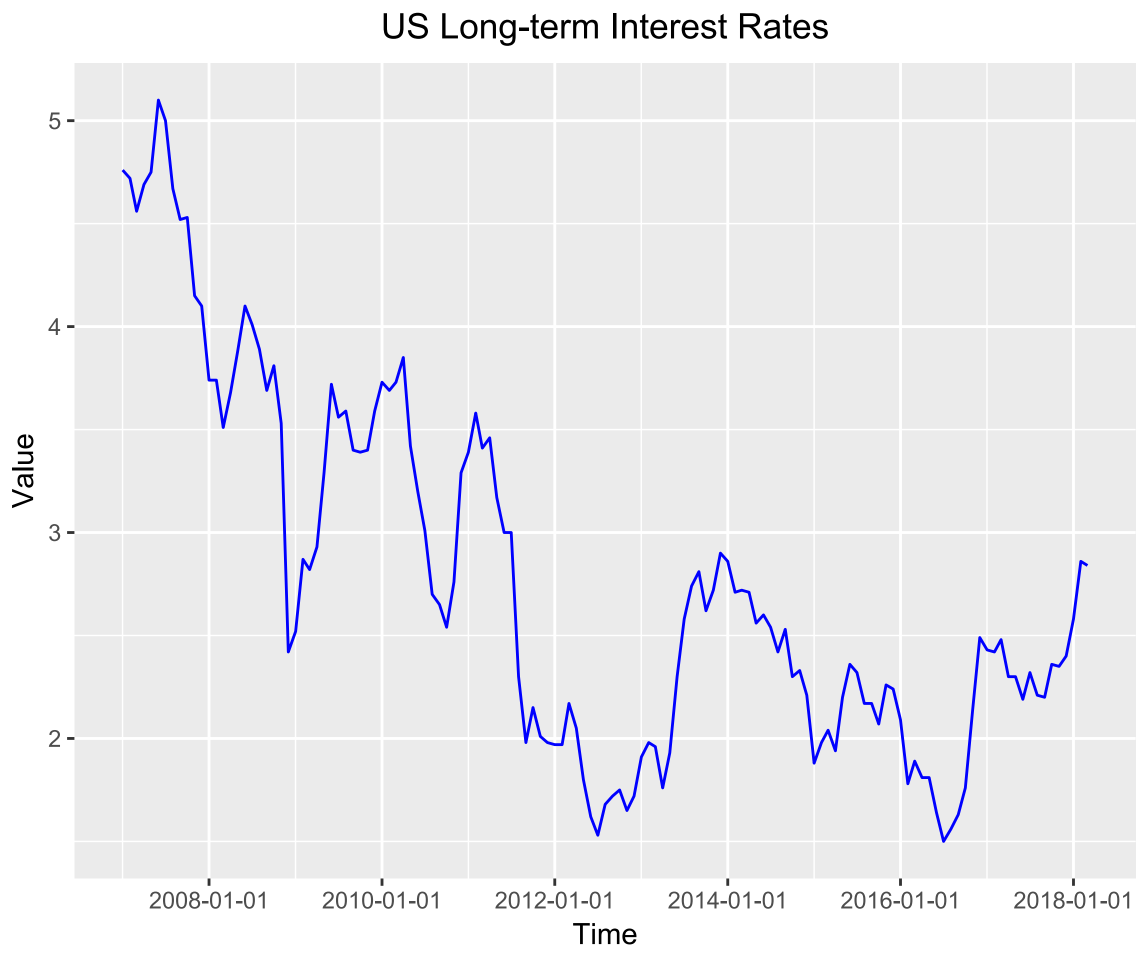
Load the neccessary libraries & the dataset

```
# Load the necessary packages
library(keras)
library(tensorflow)
```

Or install as follows:

```
devtools::install_github("rstudio/keras")
# then install Tensorflow backend as follows:
library(keras)
install_keras()
```

We will use the U.S. long-term interest rates data, available here This is a monthly data from Jan 2007 to March 2018.



First five observations

```
## [1] 4.76 4.72 4.56 4.69 4.75 5.10
```

Data preparation

Transform data to stationary

This is done by getting the difference between two consecutive values in the series. This transformation, commonly known as differencing, removes components in the data that are time dependent. Furthermore, it is easier to model using the difference, rather than the raw values, and the resulting model has a higher predictive power.

```
# transform data to stationarity
diffed = diff(Series, differences = 1)
head(diffed)
```

```
## [1] -0.04 -0.16  0.13  0.06  0.35 -0.10
```

Lagged dataset

LSTM expects the data to be in a supervised learning mode. That is, having a target variable Y and predictor X. To achieve this, we transform the series by lagging the series and have the value at time $(t-k)$ as the input and value at time t as the ouput, for a k-step lagged dataset.

```
lag_transform <- function(x, k= 1){

      lagged =  c(rep(NA, k), x[1:(length(x)-k)])
      DF = as.data.frame(cbind(lagged, x))
      colnames(DF) <- c( paste0('x-', k), 'x')
      DF[is.na(DF)] <- 0
      return(DF)
}
supervised = lag_transform(diffed, 1)
head(supervised)
```

```
##     x-1     x
## 1  0.00 -0.04
## 2 -0.04 -0.16
## 3 -0.16  0.13
## 4  0.13  0.06
## 5  0.06  0.35
## 6  0.35 -0.10
```

Split dataset into training and testing sets

Unlike in most analysis where training and testing data sets are randomly sampled, with time series data the order of the observations does matter. The following code split the **first** 70% of the series as training set and the remaining 30% as test set.

```
## split into train and test sets

N = nrow(supervised)
n = round(N *0.7, digits = 0)
train = supervised[1:n, ]
test  = supervised[(n+1):N,  ]
```

Normalize the data

Just like in any other neural network model, we rescale the input data X to the range of the activation function. As shown earlier, the default activation function for LSTM is sigmoid function whose range is [-1, 1]. The code below will help in this transformation. Note that the min and max values of the training data set are the scaling coefficients used to scale both the training and testing data sets as well as the predicted values. This ensures that the min and max values of the test data do not influence the model.

```
## scale data
scale_data = function(train, test, feature_range = c(0, 1)) {
  x = train
  fr_min = feature_range[1]
  fr_max = feature_range[2]
  std_train = ((x - min(x) ) / (max(x) - min(x)  ))
  std_test  = ((test - min(x) ) / (max(x) - min(x)  ))

  scaled_train = std_train *(fr_max -fr_min) + fr_min
  scaled_test = std_test *(fr_max -fr_min) + fr_min

  return( list(scaled_train = as.vector(scaled_train), scaled_test = as.vector(scaled_test) ,s

}



Scaled = scale_data(train, test, c(-1, 1))

y_train = Scaled$scaled_train[, 2]
x_train = Scaled$scaled_train[, 1]

y_test = Scaled$scaled_test[, 2]
x_test = Scaled$scaled_test[, 1]
```

The following code will be required to revert the predicted values to the original scale.

```
## inverse-transform
invert_scaling = function(scaled, scaler, feature_range = c(0, 1)){
  min = scaler[1]
  max = scaler[2]
  t = length(scaled)
  mins = feature_range[1]
  maxs = feature_range[2]
  inverted_dfs = numeric(t)

  for( i in 1:t){
    X = (scaled[i]- mins)/(maxs - mins)
    rawValues = X *(max - min) + min
    inverted_dfs[i] <- rawValues
  }
  return(inverted_dfs)
}
```

Modeling

Define the model

We set the argument *stateful*= TRUE so that the internal states obtained after processing a batch of samples are reused as initial states for the samples of the next batch. Since the network is stateful, we have to provide the input batch in 3-dimensional array of the form [*samples*, *timesteps*, *features*] from the current [*samples*, *features*], where:

- Samples: Number of observations in each batch, also known as the batch size.
- Timesteps: Separate time steps for a given observations. In this example the timesteps = 1
- Features: For a univariate case, like in this example, the features = 1

The batch size must be a common factor of sizes of both the training and testing samples. 1 is always a sure bet. A nice explanation of LSTM input can be found here

```
# Reshape the input to 3-dim
dim(x_train) <- c(length(x_train), 1, 1)

# specify required arguments
X_shape2 = dim(x_train)[2]
X_shape3 = dim(x_train)[3]
batch_size = 1                # must be a common factor of both the train and test samples
units = 1                     # can adjust this, in model tuninig phase

#=========================================================================================

model <- keras_model_sequential()
model%>%
  layer_lstm(units, batch_input_shape = c(batch_size, X_shape2, X_shape3), stateful= TRUE)%>%
  layer_dense(units = 1)
```

Compile the model

Here I have specified the *mean_squared_error* as the loss function, Adaptive Monument Estimation (ADAM) as the optimization algorithm and learning rate and learning rate decay over each update. Finaly, I have used the *accuracy* as the metric to assess the model performance.

```
model %>% compile(
  loss = 'mean_squared_error',
  optimizer = optimizer_adam( lr= 0.02, decay = 1e-6 ),
  metrics = c('accuracy')
)
```

Model summary

```
summary(model)

## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## lstm_1 (LSTM)                    (1, 1)                        12
## ___________________________________________________________________________
## dense_1 (Dense)                  (1, 1)                        2
## ===========================================================================
## Total params: 14
## Trainable params: 14
## Non-trainable params: 0
## ___________________________________________________________________________
```

Fit the model

We set the argument *shuffle* = FALSE to avoid shuffling the training set and maintain the dependencies between x_i and x_{i+t}. LSTM also requires resetting of the network state after each epoch. To achive this we run a loop over epochs where within each epoch we fit the model and reset the states via the argument *reset_states()*.

```
Epochs = 50
for(i in 1:Epochs ){
  model %>% fit(x_train, y_train, epochs=1, batch_size=batch_size, verbose=1, shuffle=FALSE)
  model %>% reset_states()
}
```

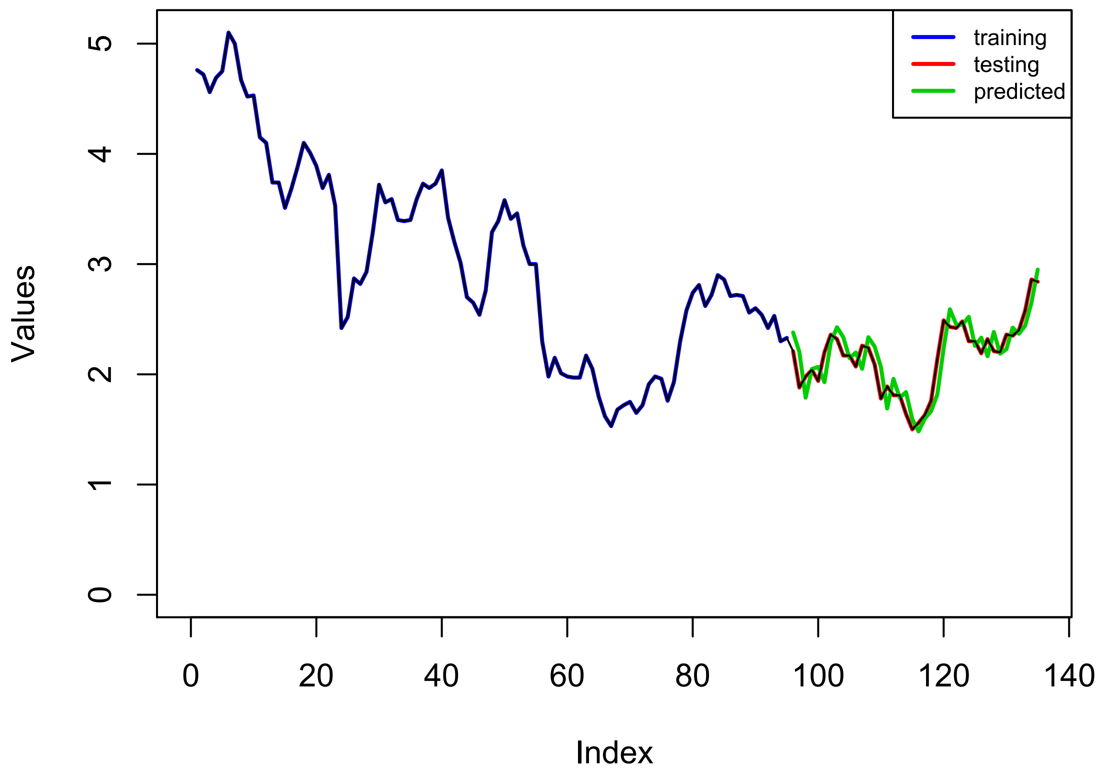
Make predictions

```
L = length(x_test)
scaler = Scaled$scaler
predictions = numeric(L)

for(i in 1:L){
     X = x_test[i]
     dim(X) = c(1,1,1)
     yhat = model %>% predict(X, batch_size=batch_size)
     # invert scaling
```

```
      yhat = invert_scaling(yhat, scaler,  c(-1, 1))
      # invert differencing
      yhat  = yhat + Series[(n+i)]
      # store
      predictions[i] <- yhat
 }
```

Plot the values



```
## RMSE:  0.01
```

Get the entire code in my git repo

Share

OLDER

Overview of Machine Learning Algorithms

NEWER

Generative Adversarial Networks

RECENTS

DEEP LEARNING
DEPLOY DEEP LEARNING MODELS WITH FLASK
2018-10-11

DEEP LEARNING
GENERATIVE ADVERSARIAL NETWORKS
2018-04-14

DEEP LEARNING
TIME SERIES FORECASTING USING LSTM IN R
2018-04-05

OVERVIEW OF MACHINE LEARNING ALGORITHMS
2018-04-03